UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

NAYANE PONTE VIANA

UM SERVIÇO PARA FLEXIBILIZAÇÃO DA TARIFAÇÃO EM NUVENS DE INFRAESTRUTURA

FORTALEZA, CEARÁ

2013

NAYANE PONTE VIANA

UM SERVIÇO PARA FLEXIBILIZAÇÃO DA TARIFAÇÃO EM NUVENS DE INFRAESTRUTURA

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Área de concentração: Engenharia de Software

Orientador: Prof. Dr. Fernando Antonio de Mota Trinta

Co-Orientador: Profa. Dra. Rossana Maria de Castro Andrade

FORTALEZA, CEARÁ

2013

NAYANE PONTE VIANA

UM SERVIÇO PARA FLEXIBILIZAÇÃO DA TARIFAÇÃO EM NUVENS DE INFRAESTRUTURA

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação, da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação. Área de concentração: Engenharia de Software

Aprovada em: __/__/____

BANCA EXAMINADORA

Prof. Dr. Fernando Antonio de Mota Trinta
Universidade Federal do Ceará - UFC
Orientador

Prof. Dra. Rossana Maria de Castro Andrade
Universidade Federal do Ceará - UFC
Co-orientadora

Prof. Dr. Nabor das Chagas Mendonça
Universidade de Fortaleza - UNIFOR

Prof. Dr. Americo Tadeu Falcone Sampaio
Universidade de Fortaleza - UNIFOR

Ao Meu Esposo, meu Amigo e meu Amor, José Ricardo Mello Viana

AGRADECIMENTOS

Primeiramente, agradeço a Deus, meu Pai todo poderoso e Senhor onipotente, A Ele toda honra, toda glória e louvor eternamente. A quem eu atribuo todas as realizações e conquistas desse mestrado, a quem eu incessantemente pedi e felizmente fui agraciado.

Ao meu esposo, José Ricardo Mello Viana, meu amigo, companheiro e aliado, que nunca mediu esforços para que esse mestrado fosse finalizado. Suportou meus momentos de fraqueza e decepção, com muito amor e paciência segurando a minha mão. Ele é o baluarte seguro que Deus tem me presenteado, a quem dedico minhas alegrias e realizações nesse mestrado conquistado. Não tenho como deixar de agradece-lo eternamente, pois além de proporcionar-me felicidades, co-orientou-me, "simplesmente!"

À minha mãe, Rita Albuquerque Ponte e ao meu pai, José Jacinto Ferreira da Ponte, que me geraram e criaram, no princípio do mestrado minha ausência lamentaram. Mas sempre torceram e vibraram por minhas conquistas e realizações, sempre rezando e abençoando-me não importando as escolhas e razões.

Aos meus 7 irmãos (Gláucia, Regina, Lincoln, Cristiane, Tereza, Daniel e Liana), meus eternos amigos, que reclamaram pensando, que eu já tinha era morrido. Sem que cada um, mesmo, soubesse, a certeza desse amor sempre me fortalece.

Ao meu "*best*"orientador, Fernando Trinta agradeço: *(i)* pelo trabalho que me confiou e *(ii)* às inúmeras correções que me ajudou *(iii)* Peço desculpas pelas momentos de provação, essa é a sina na profissão.

À querida e sempre linda, minha professora e co-orientadora Rossana Andrade, a quem admiro o trabalho e sua beldade. Agradeço pela confiança e contribuições, suas críticas e elogios sempre me deram satisfações.

Aos demais da minha família, que é grande, sobrinhos, cunhados, sogro, sogra, tios, primos, Minie e todo o restante. Meu imenso agradecimento por incentivarem essa minha fase, seja um sorriso, um boa sorte ou um "já ta quase?"

Aos meu amigos, nova família que fiz no MDCC-GREat, Benedito, André, Raynara, Jefferson, Aparecida, Paulo, Márcio, Adyson, Rafael. Carla, Sandra, Rute, Jonas, Darilú, Janaina, Débora, Rodrigo, Charles, Emanuel. Aos piauienses Ismayle, Paula, Tálison, Douglas, Ítalo Linhares, e demais representantes, no MDCC, pois tem aos milhares.

Aos meus amigos daqui não posso esquecer, Synara, Nádia, Mara e André, os de The. À CET, FAETE e UFPI, faculdades que me contrataram, Aos meus alunos, que me confiaram.

Aos funcionários, dessa instituição, que sempre me trataram com valorosa atenção. Aos professores que contribuíram com esse mestrado, Danielo, Windson, Paulo Henrique, Lincoln e Carlos.

Agradeço a banca que me acompanhou, professores Américo e Nabor e a FUNCAPI

que me financiou.

Se alguém faltou, peço perdão, mas com certeza está no meu coração.

Finalizo esse agradecimento com uma poesia, não é minha, mas bom seria: Então, meus amigos de verdade venho lhes dizer que

"Há de levar comigo uma saudade tua... Hás de ficar contigo uma saudade minha...".

(Duas Almas, WAMOSY, Alceu. Poesia completa. Alves Ed.: IEL: EDIPUCRS, 1994. p.143)

“Se o Senhor não construir a nossa casa, em vão trabalharão seus construtores”

(Salmo 126)

RESUMO

A computação em nuvem surgiu em 2006 com a ideia de transformar a computação em um serviço utilitário. Esse novo paradigma é baseado em várias tecnologias que vieram amadurecendo ao longo dos tempos, como o sistema distribuído, a computação em grade e a virtualização. Além disso, a computação em nuvem tem suas características peculiares, como a customização, a elasticidade e o pagamento por uso do serviço. Portanto, por ser um paradigma novo, a nuvem possui muitas questões em aberto que precisam ser amadurecidas, como a segurança, a disponibilidade e a tarifação.

A tarifação é uma das principais características da computação em nuvem. Nesse paradigma, o cliente paga pelo que utiliza, modelo conhecido como *pey per use*. Para isso, é preciso realizar o monitoramento do uso dos recursos a fim de tarifar de acordo com sua utilização. As provedoras de nuvem disponibilizam diferentes tipos de serviços aos seus usuários, os principais são:*(i)* Software como Serviço, *(ii)* Plataforma como Serviços e *(iii)* Infraestrutura como Serviço (hardware). No caso dos serviços disponibilizados por nuvens de infraestrutura, é necessário medir o uso dos recursos de hardware na nuvem. Porém, muitos trabalhos acadêmicos mostram que o modelo de tarifação das provedoras de nuvem não levam em consideração requisitos importantes para o cálculo da fatura do cliente.

Baseado nisso, este trabalho tem por objetivo melhorar a flexibilidade da tarifação em nuvens de infraestrutura. Para isso, ele propõe uma arquitetura e um serviço de tarifação, o *aCCountS (Cloud aCCounting Service)* e uma linguagem de domínio específico (DSL) para definição de políticas de tarifação em nuvens, chamada de *aCCountS-DSL*. Inicialmente, foi realizado um estudo na academia e na indústria a fim de coletar e classificar os requisitos para flexibilizar a cobrança de serviços na computação em nuvem e criar um novo modelo de tarifação.

A partir desses estudos foram definidos *(i)* a linguagem de tarifação proposta, os requisitos que são suportados pela linguagem, em seguida a *(ii)* a arquitetura do serviço e a descrição de suas partes fundamentais e então, *(iii)* o serviço de tarifação proposto.

Por fim, esse trabalho descreve as avaliações experimentais realizadas para testar a corretude do serviço e da linguagem propostos. Para isso, foram feitos testes reais a partir da implantação do serviço em nuvens comerciais com o objetivo de testar o *aCCountS* e a *aCCountS-DSL*.

Palavras-chave: Computação em Nuvem. Tarifação em Computação em Nuvem. Nuvens de infraestrutura.

ABSTRACT

Cloud computing has emerged in 2006 with the idea to make the utility computing service. This new paradigm is based on several mature technologies that have come to the right time, such as grid computing, distributed systems and virtualization. However, cloud computing has its peculiar features, like customization, elasticity and pay-per-use service and, moreover, to be a new paradigm, has many open questions that needs to be mature, such as security, availability and charging.

The pricing is a key feature of cloud computing. In this approach the customer pay for your use, model known as pey per use. Then, perform the monitoring of the resources use is needed in order to pricing. In cloud services infrastructure case, monitoring the hardware resources use is necessary. However, many academic studies show that the form of charging by cloud providers do not take into account important requirements for the customers invoice calculation.

Based on this, this work aims to improve the flexibility of charging for infrastructure clouds. For this, an academic and industry study was made to collect and sort the requirements for flexible charging services in cloud computing. Thus, an architecture and service charges, the aCCountS (a Cloud Accounting Service) and a domain specific language (DSL) for defining pricing policies in clouds, called Accounts-DSL, was proposed.

In this study are defined: *(i)* the service architecture and the description of its key parts, *(ii)* the accounting service proposed and how it was developed, and *(iii)* the charging language and its requirements and grammar.

Finally, the experimental evaluation performed to test the correctness of the service and the language proposed are described. For this, real test was made with the service deployment in commercial clouds in order to test the aCCountS and aCCountS-DSL features.

Keywords: Cloud Computing. Billing. Infraestructure Clouds.

LISTA DE FIGURAS

Figura 2.1 Tabela de preços de recursos da *Amazon* 26

Figura 2.2 Tabela de descontos por volume de instâncias da *Amazon* 27

Figura 2.3 Fluxo de tarifação, adaptado de Ruiz-Agundez *et al* (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011) 28

Figura 2.4 Modelo arquitetural de faturamento para o *RESERVOIR* (ELMROTH et al., 2009). 30

Figura 2.5 Algoritmos de (a) precificação e (b) cobrança para tarifação em nuvem (NARAYAN et al., 2012) 31

Figura 2.6 Serviço de precificação proposto por Caracas e Altmann(CARACAS; ALTMANN, 2007) 33

Figura 3.1 Visão Geral da Proposta. 36

Figura 3.2 Fluxo do *aCCountS* baseado no fluxo proposto por Ruiz-Agundez (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011). 37

Figura 3.3 Fluxo de tarifação no agente. 38

Figura 3.4 Diagrama de componentes UML com os principais elementos de *aCCountS*. 40

Figura 3.5 Diagrama de atividades do processo do *aCCountS* 41

Figura 3.6 Diagrama de classes do *aCCountS-Agent* 43

Figura 3.7 Diagrama de sequência de requisição das regras de medição pelo *aCCountS-Agent* para *aCCountS-Service* 44

Figura 3.8 Diagrama de sequência do fluxo de medição e mediação no *aCCountS-Agent* 44

Figura 3.9 Diagrama de sequência de envio dos dados do *aCCountS-Agent* para o *aCCountS-Service* 45

Figura 3.10 Diagrama de atividades do processo do *aCCountS* 47

Figura 3.11 Diagrama de classes do *aCCountS-Service* 48

Figura 3.12 Diagrama de sequência do *aCCountS-Service* 49

Figura 3.13 Diagrama de atividades do processo do *aCCountS* 52

Figura 4.1 Arquitetura do protótipo do *aCCountS* 63

Figura 4.2 Protótipo da arquitetura *aCCountS* - Tela inicial 68

Figura 4.3 Protótipo da arquitetura *aCCountS* - Tela de variáveis 68

Figura 4.4 Protótipo da arquitetura *aCCountS* - Cadastro de variável 69

Figura 4.5 Protótipo da arquitetura *aCCountS* - Tela de perfis 69

Figura 4.6 Protótipo da arquitetura *aCCountS* - Criar perfil 70

Figura 4.7 Protótipo da arquitetura *aCCountS* - Tela de políticas 71

Figura 4.8 Protótipo da arquitetura *aCCountS* - Cadastro de política 71

Figura 4.9 Protótipo da arquitetura *aCCountS* - Política com erros 72

Figura 4.10 Protótipo da arquitetura *aCCountS* - Visualizar política 72

Figura 4.11 Protótipo da arquitetura *aCCountS* - Trecho Código *Ruby* completo com códigos herdados 73

Figura 4.12 Protótipo da arquitetura *aCCountS* - associação entre clientes, perfis e políti-

cas 73

Figura 4.13 Protótipo da arquitetura *aCCountS* - Registros de cobrança 74

Figura 4.14 Protótipo da arquitetura *aCCountS* - dados de uso recebidos 74

LISTA DE TABELAS

Tabela 2.1 Comparação das propostas de tarifação existentes 33

Tabela 5.1 Resultados dos experimentos realizados na *Amazon EC2* e *Windows Azure* . 84

Tabela 5.2 Registros fixos usados como casos de teste 91

Tabela 5.3 Resultados obtidos no experimento 91

SUMÁRIO

1 INTRODUÇÃO 16
1.1 Motivação 19
1.2 Proposta 20
1.3 Objetivos 21
1.4 Estrutura da Dissertação 21
2 TARIFAÇÃO EM COMPUTAÇÃO EM NUVEM 22
2.1 Tarifação na Indústria 22
2.1.1 Tarifação na *Amazon EC2* 23
2.2 Trabalhos acadêmicos 27
2.3 Discussão 32
2.4 Conclusão 33
3 ACCOUNTS 35
3.1 Visão arquitetural do *aCCountS* 39
3.1.1 APIs de comunicação entre os macro-componentes do *aCCounstS* 42
3.1.2 O Agente - *aCCountS-Agent* 42
3.1.3 O Serviço - *aCCountS-Service* 46
3.1.4 A DSL de tarifação - *aCCountS-DSL* 53
3.1.5 Conclusão 61
4 O PROTÓTIPO DO *ACCOUNTS* 63
4.1 Arquitetura completa do protótipo 63
4.2 Implementação da aCCountS-DSL 64
4.3 O Compilador da DSL (*DSLCompiler*) 64
4.4 O protótipo do serviço aCCountS 66
4.4.1 Tela inicial do serviço aCCountS 67
4.4.2 Cadastro de variáveis (recursos a serem medidos) 67
4.4.3 Cadastro de Perfis 68
4.4.4 Cadastro de Políticas 70
4.4.5 Cadastro de Máquinas virtuais 72

4.4.6 Envio de dados ao agente . . . 73
4.4.7 Recebimento de dados do agente . . . 75
4.5 O protótipo do aCCountS-Agent . . . 75
4.5.1 Coleta de recursos a serem medidos . . . 76
4.5.2 Medição . . . 76
4.5.3 Mediação . . . 77
4.5.4 Envio de dados de uso . . . 78
4.5.5 Temporização do agente . . . 79
4.5.6 Conclusões . . . 79
5 AVALIAÇÃO EXPERIMENTAL . . . 81
5.1 Avaliação dos protótipos . . . 81
5.1.1 Avaliação do *aCCountS* . . . 81
5.1.1.1 Política *simplesUsoSobPos* . . . 82
5.1.1.2 Política *simplesTempoSobPos* . . . 82
5.1.1.3 Política *simplesUsoRes1Pos* . . . 83
5.1.1.4 Resultados do Experimento . . . 83
5.1.2 Avaliação da *aCCountS-DSL* . . . 84
5.1.2.1 Política *SobMedUsoPosPlus* . . . 85
5.1.2.2 Política *Res1MedUsoPosPlus* . . . 86
5.1.2.3 Política *SobPeqUsoPosPlus* . . . 87
5.1.2.4 Política *SobMedTempoPosPlus* . . . 87
5.1.2.5 Política *SobMedUsoPrePlus* . . . 88
5.1.2.6 Política *SobMedUsoPos* . . . 88
5.1.2.7 Política *Res1PeqUsoPosPlus* . . . 89
5.1.2.8 Política *Res1MedTempoPosPlus* . . . 89
5.1.2.9 Política *Res1MedUsoPrePlus* . . . 90
5.1.2.10 Política *Res1MedUsoPos* . . . 90
5.1.2.11 Realização do Experimento . . . 91
5.1.3 Conclusões . . . 92
6 CONCLUSÕES E TRABALHOS FUTUROS . . . 93

REFERÊNCIAS BIBLIOGRÁFICAS 95
APÊNDICE A – GRAMÁTICA DA *ACCOUNTS-DSL* 98
APÊNDICE B – GERADOR DE CÓDIGO DO *DSLCOMPILER* 103

1 INTRODUÇÃO

Desde a década de 60, cientistas já pensavam na ideia da computação utilitária. Na década de 60, Joseph Carl Robnett Licklider, um dos responsáveis pelo desenvolvimento da *ARPANET* (*Advanced Research Projects Agency Network*) já havia introduzido a ideia de uma rede de computadores intergalática (LICKLIDE, 2012) e, ainda na década de 60, John McCarthy, um cientista norte-americano da área de inteligência artificial, propôs a ideia de que a computação deveria ser organizada na forma de um serviço de utilidade pública, em que uma agência de serviços disponibilizaria e cobraria uma taxa pelo uso de serviços de computação (CHIRIGATI, 2012). Esses pesquisadores já vislumbravam a ideia que no futuro a computação seria adquirida de uma forma parecida com a energia elétrica. Apesar da computação utilitária não ter avançado após os anos seguintes, a sua proposição, ideias semelhantes foram revisitadas recentemente em um novo conceito chamado de *Cloud Computing* (Computação em Nuvem, tradução livre).

Esse termo relaciona-se com um novo paradigma para a computação, onde busca-se deslocar toda ou parte de uma infraestrutura computacional para a rede. Porém, na realidade, a rede apresenta-se como meio de acesso a recursos que variam desde aplicações, hardware ou espaço de armazenamento de dados. O termo foi introduzido em 2006, quando em uma conferência sobre "Estratégias de Motores de Pesquisa", o termo *"Cloud Computing"* foi mencionado pelo CEO da *Google*, Eric Schmidt, indicando que a empresa utilizaria tal expressão para seu novo modelo de negócios. Este modelo permitiria acesso ubíquo a dados e computação que se localizariam em uma "nuvem" (*"cloud"*) de vários servidores, localizados em um centro de dados remoto. No mesmo ano, a *Amazon.com*, uma das maiores lojas de comércio eletrônico da Internet, anunciou um pioneiro e dos mais importantes serviços no modelo de computação em nuvem, serviços em *Cloud Computing*, até os dias de hoje: o *Elastic Compute Cloud* (EC2), como parte do *Amazon Web Services* (AMAZON, 2013). A partir de então, o termo "Computação em Nuvem" passou a ganhar mais espaço e outras empresas também começaram a investir nessa área como a *IBM* e a *Microsoft*.

Com o sucesso da Computação em Nuvem, muitos estudos (BUYYA et al., 2009; VAQUERO et al., 2008; ARMBRUST et al., 2009; MELL; GRANCE, 2011; SOTOMAYOR et al., 2009) surgiram nessa área e com eles muitas definições para essa nova tecnologia. Este trabalho de dissertação baseia-se na definição proposta por Vaquero *et al.* (VAQUERO et al., 2008), em que os autores afirmam:

> "*A nuvem é um grande reservatório de recursos virtualizados facilmente utilizáveis e acessíveis (como hardware, plataformas de desenvolvimento e/ou serviços). Esses recursos podem ser dinamicamente reconfigurados para ajustar a carga (escala) variável do sistema, permitindo também um uso ótimo dos recursos. Tal reservatório é geralmente explorado por um modelo pay-per-use no qual as garantias são oferecidas por um Provedor de Infraestrutura por meio de SLAs (Service Level Agreement - Acordo de Nível de Serviço)*".

Esses serviços são implantados na nuvem e disponibilizados por meio da internet. A literatura sobre o tema, como apresentado por Buyya el al. (BUYYA; BROBERG; GOSCINSKI, 2011) apresenta três modelos de entrega de serviços que podem ser vistos como sobreposições em diferentes níveis da pilha de uma infraestrutura de TI. No nível mais alto estão as aplicações, ou *Software como Serviço (SaaS - do inglês, Software as a Service)*, no qual clientes podem utilizar sistemas de software com fins específicos. A *Salesforce.com* que se baseia no modelo de SaaS, oferece aplicações de produtividade de negócios que residem totalmente em seus servidores, permitindo que clientes personalizem e acessem aplicativos sob demanda. Outros exemplos são o Google docs[1] e a Office live [2].

No nível intermediário está a Plataforma como Serviço *(PaaS - do inglês, Platform as a Service)*, que oferece tanto um ambiente de desenvolvimento quanto execução de aplicações, incluindo *frameworks*, ferramentas de desenvolvimento e testes de aplicações para nuvens específicas. Tanto no nível de SaaS, quanto no de PaaS, o usuário da nuvem não controla ou administra a infraestrutura subjacente, como rede, servidores, sistema operacional. Dessa forma, vários modelos de programação e serviços especializados (por exemplo, acesso a dados, autenticação, pagamento) são oferecidos como blocos de construção para novas aplicações. O *Google AppEngine* (GOOGLE, 2013) é um exemplo de plataforma como serviço, um ambiente escalável para desenvolvimento e hospedagem de aplicações web, escritas em uma linguagem de programação específica, tal como *Python* ou *Java* e com um serviço proprietário de armazenamento de dados. O desenvolvedor, por sua vez, tem a possibilidade de, em qualquer lugar, programar e testar os sistemas em desenvolvimento contando com os recursos necessários.

No nível mais baixo, a *Infraestrutura como Serviço (IaaS, do inglês Infrastructure as a Service*) oferece máquinas virtualizadas com sistemas operacionais próprios, onde clientes podem instalar e configurar aplicações de acordo com seus interesses. Neste último nível, o cliente da nuvem tem controle sobre as configurações das máquinas virtuais, porém não sobre a infraestrutura da nuvem. Muitas empresas tem entrado no negócio da computação em nuvem, como a *Microsoft*, com o *Windows Azure* (AZURE, 2013), *IBM*, com a *IBMSmartCloud* (IBM, 2012), entre outras. Entretanto, a maior representante é a *Amazon EC2* (LIVESTREAM, 2013b). Segundo Marten Mickos, CEO da *Eucalyptus Systems*, a API da *Amazon EC2* deveria ser a base para um padrão da indústria e comparou a *Amazon EC2*, como um padrão para a nuvem, com a *IBM*, que foi um padrão para o *PC*.

Além desses níveis e serviços, há ainda outros exemplos como a disponibilização de dados como serviço (DaaS) em que os dados são o produto e podem ser utilizados sob demanda. Vários nichos de negócio tem se aproveitado do modelo de computação para criar novas oportunidades, como Jogos (exemplo, Steam (STEAM, 2013)), Músicas (exemplos, *(i)* iTunes store (STORE, 2013) ou o aplicativo *(ii)* rdio (RDIO, 2013)) ou Vídeos sob demanda (exemplo, Netflix (NETFLIX, 2013)).

Além do modelo de entrega, nuvens podem ser também classificadas quanto ao modelo de implantação. Nesta outra classificação, uma nuvem pode ser como *(i)* pública, disponibilizada ao público em geral (ARMBRUST et al., 2009), *(ii)* privada, interno de uma organiza-

[1] http://www.google.com/google-d-s/documents/

[2] http://www.microsoft.com/online/pt-br/office-live-meeting.aspx

ção empresarial ou outro, não disponível para o público em geral (ARMBRUST et al., 2009), *(iii)* comunitária, partilhada por várias organizações e suporta uma comunidade específica que divide as mesmas preocupações (como por exemplo, a missão, os requisitos de segurança, as considerações políticas e de conformidade (MELL; GRANCE, 2009) e *(iv)* híbrida, toma forma quando uma nuvem privada é complementada com capacidade de computação de nuvens públicas (SOTOMAYOR et al., 2009).

De acordo com os modelos existentes, a computação em nuvem possibilita uma série de cenários interessantes de aplicação. Por exemplo, no caso de um SaaS, clientes da nuvem podem utilizar aplicativos a partir de qualquer dispositivo, seja ele um computador pessoal, um notebook, um tablet ou um smartphone interligados na internet.

Já os serviços de infraestrutura permitem que seus clientes possam iniciar seus negócios com baixo investimento financeiro, contar com uma infraestrutura elástica e pagar apenas pelo que utilizar. Posteriormente, eles podes aumentar seu poder computacional na nuvem, crescendo a medida que for sendo necessário e possível. Enquanto isso, as grandes empresas podem migrar seus centros computacionais para essas provedoras de nuvem de infraestrutura, transferindo as responsabilidades de manutenção das máquinas e gerenciando-as como se estivessem localmente.

Além disso, os clientes da nuvem podem aumentar ou diminuir o consumo de recursos, conforme sua necessidade, lhes parecendo serem infinitos. Isso acontece porque a nuvem é construída por uma grande quantidade de computadores robustos interligados, que, por meio da característica elástica da nuvem, pode aumentar ou diminuir a disponibilidade de recursos para as aplicações dos clientes, dependendo da demanda por esses serviços.

A grande utilização de recursos físicos nas nuvens gera um grande consumo de energia elétrica, entretanto, existem muitos estudos nessa área que orientam como configurar as máquinas virtuais para que mantenham um bom desempenho e economize energia, como o trabalho de Imada (IMADA; SATO; KIMURA, 2009). Esse trabalha realiza uma avaliação experimental de quais configurações de máquinas virtuais e modelos de migrações gastam menos energia sem comprometer o desempenho do serviço.

Apesar das vantagens apresentadas, o modelo de computação em nuvem ainda tem uma série de questões em aberto. Uma delas, senão a considerada mais complexa, relaciona-se à segurança das informações e aplicações. Muitas empresas não confiam em migrar seus dados pessoais ou de suas organizações para computadores de terceiros, os quais podem ser acessados por pessoas mal intencionadas. Outra questão é a disponibilidade, pois serviços e dados na nuvem são acessados por meio de uma rede. Isso gera um ponto de falha, uma vez que quando a conexão for perdida, usuário perde acesso aos serviços.

Existem vários trabalhos que procuram caracterizar as peculiaridades da Computação em Nuvem. Segundo Buyya *et al.* (BUYYA; BROBERG; GOSCINSKI, 2011) as principais características da nuvem são quatro: *(i)* auto-atendimento (*self-service*), *(ii)* elasticidade, *(iii)* customização e *(iv)* pagamento por uso. Em relação ao primeiro, clientes de uma nuvem podem se auto servir dos recursos disponíveis na nuvem, solicitar, personalizar, pagar e usar os mesmos sem intervenção de operadores humanos. Remotamente, o usuário cria as máquinas

virtuais, implanta suas aplicações, estrutura o ambiente de programação através de APIs e ferramentas de desenvolvimento. Além disso, ele pode monitorar sua conta, realizar o pagamento, gerenciar as instâncias através de um ambiente próprio proporcionado pela nuvem utilizando um navegador web em seu dispositivo.

Quanto a elasticidade, a computação em nuvem possibilita a ilusão de infinitos recursos disponíveis de acordo com a demanda. Em particular, espera-se que os recursos adicionais possam ser provisionados automaticamente e de forma rápida quando a aplicação aumentar sua carga, e liberados, também de forma automática quando a carga diminuir.

A customização, para Buyya (BUYYA; BROBERG; GOSCINSKI, 2011), é a possibilidade do cliente da nuvem poder personalizar seu ambiente de acordo com suas necessidades ou preferências. Um exemplo disso é o fato dos clientes de uma nuvem optarem por utilizarem o sistema operacional Windows ao invés do Linux.

No pagamento por uso dos serviços, algumas provedoras de nuvem possuem planos que possibilitam eliminar o compromisso prévio dos usuários, permitindo pagar pelo uso dos recursos somente pelo tempo utilizado, modelo chamado de "*pay as you go*". Os serviços devem ter uma contabilização de curto prazo (por exemplo, por hora), permitindo aos utilizadores liberarem e não pagarem pelos recursos tão logo eles não sejam mais necessários. Por estas razões, as nuvens poderiam implementar recursos para permitir o comércio eficiente do serviço, tais como preços, contabilidade e faturamento para diferentes tipos de serviço (por exemplo, o armazenamento, processamento e largura de banda) e sendo relatado em tempo real, proporcionando assim uma maior transparência, fatores que motivaram a realização desse trabalho.

1.1 Motivação

Dentre as peculiaridades da computação em nuvem, esta dissertação explora questões relacionadas à tarifação dos serviços em nuvem. Muitos estudos realizados no modelo de tarifação da indústria apontam ele como inflexível (I.R.; Y.K.; P.G., 2006; CARACAS; ALTMANN, 2007; **??**), isto é, não leva em consideração muitos requisitos da computação em nuvem que poderiam estar sendo tarifados e trazer maiores diversidades de serviços aos seus clientes, como por exemplo ser tarifado pelo uso dos recursos de hardware ou obter descontos quando suas configurações proporcionasse economia de energia. A forma de pagamento depende do modelo (SaaS, PaaS ou IaaS) utilizado pelo clientes. Elas variam desde uma taxa fixa mensal ou cobrança de acordo com a quantidade de recursos consumidos (dados, instâncias, dentre outros). Estes modelos, em geral, foram primeiramente propostos pela indústria. Mais recentemente, começaram a surgir pesquisas acadêmicas relacionadas ao assunto. Estas pesquisas buscam por novas formas de cobrança para recursos de nuvem, ou mesmo *frameworks* de tarifação para nuvens (I.R.; Y.K.; P.G., 2006; ELMROTH et al., 2009; CARACAS; ALTMANN, 2007; NARAYAN et al., 2012).

Esses trabalhos buscam apontar novos critérios que poderiam ser utilizados para compor o valor final de um recurso na nuvem, bem como novas formas de pagamento, como o modelo pré-pago. Segundo Lucrédio e Silva (SILVA; LUCREDIO, Sept.), ferramentas e

mecanismos que facilitem o monitoramento e tarifação de recursos, auxiliando as funções administrativas de gerenciamento da infraestrutura da nuvem, são imprescindíveis para o sucesso da computação em nuvem.

Apesar da tarifação ser apontada como uma tarefa importante, muitas plataformas abertas, que existem para se montar uma nuvem privada, não oferecem serviços próprios de tarifação dos seus recursos, como o *OpenNebula*, o *OpenStack* e o *Eucalyptus*. Aqueles, portanto, que forem utilizar essas plataformas para fins comerciais terão que criar suas próprias soluções de cobrança.

1.2 Proposta

Devido às questões apontadas na seção 1.1, esse trabalho se propõe a desenvolver uma solução para criar um serviço de tarifação em nuvem que facilite o trabalho do administrador e permita que sejam definidas politicas flexíveis (o administrador poderá criar qualquer regra de cobrança por meio da linguagem definida na proposta) em um nível de abstração alto. Com isso pode-se utilizar recursos que não estão sendo utilizados para tarifação em outras soluções, tais como, a taxação de uso de recursos (CPU, transação no banco de dados) e a contabilização da economia de energia.

Vários autores (I.R.; Y.K.; P.G., 2006; ELMROTH et al., 2009; CARACAS; ALTMANN, 2007; NARAYAN et al., 2012) citam que existem alguns critérios como bem estar (descontos, incentivos, clareza na definição da conta), preço justo, perfil do usuário, SLA (*Service Level Agreement*), tarifação por uso dos recursos, entre outros, que não são utilizados nas definições da politica de tarifação em nuvem. O serviço proposto busca justamente unificar e uniformizar a definição dessas politicas. A definição das políticas é feita por uma linguagem de domínio específico que busca facilitar a definição das regras de cobrança de uma nuvem de infraestrutura.

Esse trabalho iniciou pelas pesquisas na área acadêmica e na área da indústria, o que permitiu criar uma linguagem de tarifação, chamada *aCCountS-DSL*, tão flexível quanto possível para atender diferentes necessidades dessas duas esferas. A partir dessa linguagem, foi possível criar uma arquitetura para a implementação de um serviço de tarifação flexível, chamado *aCCountS-Service*. Por meio dele é possível definir e contabilizar diferentes políticas de negócio. Para tornar o serviço mais flexível, ele foi construído para ser utilizado desacoplado na nuvem, podendo assim atender qualquer infraestrutura de nuvem que disponibilize serviços de IaaS, utilizando diferentes sistemas operacionais. Para monitorar as nuvens foi criado um agente, que envia os dados de consumo dos clientes, chamado *aCCuntS-Agent*. O conjunto de todos estes elementos é chamado *aCCountS*.

A proposta desta dissertação situa-se entre os trabalhos que exploram questões de tarifação em computação em nuvens. Porém, faz-se necessário enfatizar que existe uma limitação no escopo, que apesar de importante, não é considerado nesta pesquisa. A proposta *aCCountS* é voltada para nuvens de infraestrutura (IaaS). Desta forma, questões de SaaS e PaaS não são abordadas. Da mesma forma, não houve a preocupação em criar um serviço seguro e

escalável. Acredita-se que existam soluções que podem ser incorporadas à nossa proposta com intuito de tratar tais questões.

1.3 Objetivos

Este trabalho tem como objetivo *(i)* fazer uma revisão literária sobre tarifação em serviços de nuvem; *(ii)* modelar e implementar um serviço de tarifação em nuvens de infraestrutura e modelar e especificar uma DSL para definição de politicas de tarifação para nuvens de infraestrutura; e *(iii)* realizar experimentos para validar essa proposta.

1.4 Estrutura da Dissertação

Essa dissertação encontra-se dividida em cinco capítulos a partir dessa introdução. O Capítulo 2 faz uma revisão sobre a tarifação em nuvem mostrando também os principais trabalhos relacionados a essa dissertação. O Capítulo 3 apresenta a proposta, o *aCCountS*, que está subdividido na explicação do serviço, na descrição da arquitetura de tarifação e na definição da linguagem proposta, a *aCCountS-DSL*. O Capítulo 4 apresenta a avaliação empírica desse trabalho, por meio da avaliação experimental do protótipo do serviço, cujo objetivo é validá-lo em termos de corretude. Por fim, no Capítulo 5 são apresentadas as conclusões e as sugestões para trabalhos futuros.

2 TARIFAÇÃO EM COMPUTAÇÃO EM NUVEM

Conforme já explicado anteriormente, umas das principais características da computação em nuvem é o pagamento por uso dos serviços (*pay per use*). Isso permite que um cliente ou organização conte com uma infraestrutura computacional robusta, no caso dos serviços de nuvem de infraestrutura, sem um alto investimento financeiro inicial.

Neste capítulo busca-se fazer uma revisão literária sobre estudos em tarifação em nuvem, porém, não abordando apenas trabalhos acadêmicos, mas também os modelos de tarifação utilizados na indústria, pois com ela a computação em nuvem tem alcançado grandes avanços. Especificadamente, tomou-se por modelo a *Amazon EC2*, por essa ser a maior representante em serviço de nuvem de infraestrutura e as demais seguirem seu modelo de tarifação. Dessa maneira, na subseção 1.1 será abordado a tarifação na *Amazon EC2* e na subseção 1.2 os trabalhos realizados na academia.

2.1 Tarifação na Indústria

Dado que alguns serviços de tarifação estão disponíveis no mercado, tais como *JBilling* (JBILLING, 2013) e *Cloud Billing* (IBM, 2013), os mesmos foram estudados a fim de comparação com o serviço de tarifação proposto. Por meio das pesquisas realizadas, constatou-se que eles são baseados em *BRMS - Business Rule Management System* (sistemas de regras de negócios). A vantagem desse tipo de sistema é a possibilidade de definições de regras de tarifação, inclusive políticas de cobrança que levam em consideração o uso dos recursos, descontos para incentivar a economia de energia, descontos na violação do SLA e descontos de incentivo ao usuário. Entretanto, existem grandes desvantagens. Segundo o grupo de pesquisa e desenvolvimento de *software Hartmann Software Group* (GROUP, 2012), *(i)* os BRMS não são otimizados para tarefas computacionais que requerem alto processamento; *(ii)* não são ambientes ideais para escrever algoritmos complexos (diminuindo a eficiência do mecanismo de regras e tornando difícil para os usuários definirem formas mais complexas); e *(iii)* pertubações em seus modelos de objeto (ou regras) podem ter implicações em todo o sistema (dado que as regras são definidas por outras regras ou modelos de objetos, criando um caminho de dados para realizar a inferência).

Outra crítica em relação a tais sistemas é a complexidade dos mesmos, além da curva de aprendizado longa para que usuários consigam dominar a definição de suas regras de negócio. A definição de uma linguagem específica para tarifação, com recursos voltados exclusivamente para definição das regras de cobrança pode facilitar o entendimento e criação das políticas de tarifação. O serviço proposto permite basicamente a definição de operações matemáticas e lógicas sobre recursos e valores definidos de preços para tais recursos. Na visão defendida nesta proposta, o uso de uma DSL específica para tarifação seria um facilitador em detrimento do uso de BRMSs.

2.1.1 Tarifação na *Amazon EC2*

A *Amazon Elastic Compute Cloud (Amazon EC2)* é o serviço de infraestrutura computacional mais utilizado no mercado (LIVESTREAM, 2013a). Ela disponibiliza seus serviços aos usuários por meio de instâncias. Essas possuem diferentes configurações para atender as distintas necessidades e possibilidades dos clientes. Normalmente, elas são categorizadas em relação à capacidade computacional da instância e aos software disponíveis. As instâncias são configuradas na infraestrutura da nuvem através de máquinas virtuais (VMs). Por meio das VMs elas podem ser criadas, configuradas, manipuladas e utilizadas pelos clientes de acordo com as políticas de negócio de cada instância.

Assim como todo serviço em nuvem, a *Amazon* tarifa seus serviços pela utilização (*pay per use*). Cada máquina virtual instanciada por um cliente está associada a um perfil de hardware (tipo de instância) e seu custo é proporcional ao seu poder computacional. A *Amazon* contabiliza o uso das instâncias pelo tempo em que as mesmas estão ligadas, mesmo se as máquinas não estiverem sendo utilizadas. Fazendo uma comparação grosseira entre o consumo desses serviços, considere-se duas instâncias iguais, ativas em um mesmo intervalo de tempo. A primeira instância consome 100% do poder computacional da máquina e a segunda, 10% desse recurso. Ao final do processo, ambas pagarão o mesmo valor, pois a *Amazon* contabiliza o tempo em que a máquina está ativa e não o consumo de recursos de *hardware* (CPU, memória, HD). Adicionado a isso, é contabilizado cada licença de software comercial utilizado e cada serviço ou recurso consumido pelo cliente.

Resumindo, na Amazon o serviço é comercializado pelo uso de uma hora, sem contar os software, serviços e recursos que são somados a parte. O preço de uma hora de uso dos recursos da *Amazon* depende de quatro fatores. O primeiro é o tipo de contrato realizado entre o cliente e a provedora; o segundo é o perfil de máquina; o terceiro é a localização geográfica do *datacenter*; e a quarta, o sistema operacional utilizado. O primeiro requisito pode ser classificado em quatro tipos:

- *Instâncias sob demanda*, que permitem que o cliente pague pela capacidade computacional por hora, sem nenhum compromisso de longo prazo. Com isso, o cliente deixa de arcar com o custo e as complexidades do planejamento, aquisição e manutenção de hardware, em geral, reduzindo seus custos (AMAZON, 2013);

- *Instâncias reservadas*, que oferecem a opção de um pagamento único e acessível para cada instância que o cliente reservar, que em troca, recebe um desconto significativo sobre a taxa por hora para essa instância. Existem três tipos de instância reservada (instâncias reservadas de utilização leve, média e pesada) que permitem equilibrar o valor inicial pago e o preço efetivo da hora (AMAZON, 2013);

- *Instâncias spot*, que permitem que clientes possam negociar o uso de capacidade não normalmente utilizada na nuvem. As instâncias são cobradas pelo preço *spot*, que é definido pelo *Amazon EC2* e oscila periodicamente em função da oferta e da demanda por capacidade desse tipo de instância. Para utilizar instâncias *spot*, o cliente faz requisição por instâncias deste tipo, com parâmetros que indicam a zona de disponibilidade desejada,

o número de instâncias *spot* que deseja executar e o preço máximo que ele está disposto a pagar por hora de instância. Se a oferta de preço máximo do cliente exceder o preço *spot* atual, sua requisição será atendida e suas instâncias serão executadas até que ele opte por encerrá-las ou até que o preço *spot* exceda o seu preço máximo (o que ocorrer primeiro) (AMAZON, 2013).

O segundo fator utilizado para definir os preços dos recursos é o tipo de instância utilizada, ou seja, o perfil de hardware disponível para a máquina virtual, que são:

- *Instâncias padrão*: De modo geral esse tipo de instâncias fornecem aos clientes um conjunto equilibrado de recursos e uma plataforma de baixo custo adequada para uma ampla variedade de aplicativos.
- *Micro Instâncias*: Essas oferecem uma pequena quantidade de recursos consistentes da CPU e permitem que o cliente aumente por um curto período de tempo esta capacidade da CPU quando demandas pelo serviço (AMAZON, 2013);
- *Instâncias de mais memória*: Estas instâncias oferecem grandes tamanhos de memória para aplicativos de alta taxa de transferência, incluindo banco de dados e aplicativos de cache de memória (AMAZON, 2013);
- *Instâncias de CPU de alta performance*: Ela têm proporcionalmente mais recursos de CPU do que memória (RAM) e são adequadas para aplicativos com processamento intensivo (AMAZON, 2013);
- *Instâncias de computação em cluster*: Estas instâncias oferecem proporcionalmente CPU de alta performance, ou seja, desempenho de rede maior e são bastante adequadas para aplicativos de Computação de Alta Performance (HPC - High Performance Computer) e outros aplicativos exigentes relacionados à rede (AMAZON, 2013);
- *Instâncias de cluster com mais memória*:Estas instâncias oferecem recursos de memória e CPU proporcionalmente elevados, com maior performance de rede, e são bem adequados para aplicativos com uso intensivo de memória, incluindo análise em memória, análise em gráfico e computação científica (AMAZON, 2013);
- *Instâncias de cluster GPU*: Estas instâncias unidades de processamento gráficos de uso gerais (GPUs - Grafic Processing Unit ou Unidade de Processamento Gráfico) de unidades de processamento com, proporcionalmente, alta utilização de CPU e maior desempenho de rede para aplicativos, beneficiando-se de processamento altamente em paralelo, incluindo HPC, renderização e aplicativos de processamento de mídia (AMAZON, 2013);
- *Instâncias de E/S elevada*: Estas instâncias oferecem desempenho muito elevado de E/S para disco. Elas são idealmente adequadas para cargas de trabalho de banco de dados com desempenho muito elevado (AMAZON, 2013);

- *Instâncias de armazenamento de alta capacidade*: Estas instâncias oferecem densidade de armazenamento por instância proporcionalmente superior, e são idealmente adequadas para aplicativos que se beneficiam de performance de E/S sequencial elevada em conjuntos de dados muito grandes (AMAZON, 2013);

Em relação ao terceiro elemento de composição de preço para a *Amazon*, os vários *datacenters* que abrigam a nuvem de recursos encontram-se em diferentes regiões da terra. Estes diferentes *datacenters* possuem diferentes características de disponibilidade e acesso. Além disso, operam em diferentes países, o que consequentemente, gera custos diferentes para manutenção de sua operação. A *Amazon* possui zonas de disponibilidade em: *(i)* Leste dos Estados Unidos (Norte da Virgínia); *(ii)* Oeste dos Estados Unidos (Oregon); *(iii)* Oeste dos Estados Unidos (Norte da Califórnia); *(iv)* União Europeia (Irlanda); *(iv)* Ásia-Pacífico (Cingapura); *(v)* Ásia-Pacífico (Tóquio); *(vi)* Ásia-Pacífico (Sydney) e *(vii)* América do Sul (São Paulo).

O quarto requisito é o sistema operacional utilizado, que pode ser: *(i)* Red Hat Enterprise Linux; *(ii)* Windows Server; *(iii)* Oracle Enterprise Linux; *(iv)* SUSE Linux Enterprise; *(v)* Amazon Linux AMI; *(vi)* Ubuntu; *(vii)* Fedora; *(viii)* Gentoo Linux e *(ix)* Debian. Como alguns destes sistemas são proprietários, seus custos de licenciamento recaem sobre os valores das máquinas virtuais que os utilizam.

A Figura 2.1 ilustra a definição do preço baseado nessas quatro questões. O primeiro requisito, tipo de contrato é *on demand* (sob-demanda). O segundo requisito, perfil de *hardware*, estão listados os cinco tipos principais de instâncias: padrão, padrão de segunda geração, microinstâncias, mais memória e CPU de alta performance. O terceiro requisito, localização do *datacenter*, é São Paulo. Por fim, o quarto requisito, sistema operacional, é o Linux.

A Amazon também disponibiliza software gratuitos e comerciais para atender a demanda dos clientes, dentre eles existem os (*i) bancos de dados*, *(ii)* Servidores de aplicativos, *(iii)* Gerenciadores de conteúdo e *(iv)* Software para inteligência de negócios (AMAZON, 2013).

Além de software, são ofertados serviços e recursos adicionais que facilitam a configuração e manutenção das instâncias. Como exemplos de serviços tem-se: *(i)* Flexibilidade; *(ii)* Elasticidade; *(iii)* Controle; *(iv)* Uso com outros *Amazon Web Services*; *(v)* Substituição de instâncias pode ser rápida e previamente encomendada; *(vi)* Serviço de segurança; e *(vii)* Serviço para inicialização fácil. Tais serviços facilitam a utilização da nuvem, por exemplo, o serviço de elasticidade, que permite ao cliente aumentar ou diminuir sua capacidade computacional em minutos, possibilitando-o criar simultaneamente uma, centenas ou até milhares de instâncias no servidor.

Como exemplo de recursos, tem-se: *(i)* Amazon Elastic Block Store (EBS); *(ii)* Instâncias otimizadas para EBS; *(iii)* Vários locais; *(iv)* Endereços *Elastic IP*; *(v)* Amazon Virtual Private Cloud; *(vi)* Amazon CloudWatch; *(vii) Auto Scaling*; *(viii) Elastic Load Balancing*; *(viii) High Performance Computing (HPC) Clusters*; *(ix)* Instâncias de E/S elevada; *(x)* Instâncias com alta capacidade de armazenamento; *(xi) VM Import/Export* e *(xii) AWS Marketplace*.

Preços das instâncias on demand

Região: América do Sul (São Paulo)

	Uso do Linux/UNIX
Instâncias on demand padrão	
Pequena (padrão)	$0.080 por hora
Médio	$0.160 por hora
Grande	$0.320 por hora
Extragrande	$0.640 por hora
Instâncias on demand padrão de segunda geração	
Extragrande	$0.680 por hora
Dupla extragrande	$1.360 por hora
Microinstâncias on demand	
Micro	$0.027 por hora
Instâncias on demand com mais memória	
Extragrande	$0.540 por hora
Dupla extragrande	$1.080 por hora
Quádrupla extragrande	$2.160 por hora
Instâncias on demand com CPU de alta performance	
Médio	$0.200 por hora
Extragrande	$0.800 por hora

Figura 2.1: Tabela de preços de recursos da *Amazon*

Esses recursos permitem usufruir melhor das características que a nuvem oferece, como o recurso *Amazon Elastic Block Store*, que oferece armazenamento persistente para as instâncias do *Amazon EC2*.

Dessa maneira, o cliente escolhe um pacote com os quatro requisitos básicos (tipo de contrato, perfil de *hardware*, localização do *datacenter* e sistema operacional), que identificará o preço do serviço por hora, ou seja, sua política de negócio, base para contabilizar o uso da infraestrutura da *Amazon* e, em seguida, seleciona-se os software, recursos e serviços adicionais, os quais seus valores serão somados à conta do cliente.

Além disso, a *Amazon* tem uma política de descontos por volume de instância reservada. Quando um cliente adquiri um número suficiente de instâncias reservadas em uma região da AWS, ele recebe automaticamente descontos em suas taxas iniciais e taxas por uso para futuras compras de instâncias reservadas nessa região da AWS. Os níveis de instâncias reservadas são determinados com base no preço sem desconto das taxas iniciais das instâncias reservadas ativas que se tem em cada região da AWS. A Figura 2.2 ilustra a tabela de descontos por volume de instâncias da *Amazon*.

Se a quantidade de reservas de um cliente atingir o valor entre 250.000 USD (U.S Dólares), e 2.0000.000 USD, ele obterá um desconto de 10% na fatura, se o consumo aumentar para um valor entre 2.000.000 USD e 5.000.000 USD, o desconto será de 20% e se atingir um valor maior que 5.000.000 USD, o desconto será ainda maior e combinado diretamente com a administração da *Amazon*.

Descontos por volume de instância reservada		
Total de instâncias reservadas	**Desconto inicial**	**Desconto por hora**
Menos de 250.000 USD	0%	0%
250.000 a 2.000.000 USD	10%	10%
2.000.000 a 5.000.000 USD	20%	20%
Mais de 5.000.000 USD	Entre em contato conosco	Entre em contato conosco

Figura 2.2: Tabela de descontos por volume de instâncias da *Amazon*

A maior questão em relação a esse modelo de política de tarifação é a sua cobrança por tempo de utilização, não considerando a quantidade de recursos utilizados, dado que um cliente que utiliza menos recursos (por motivos adversos) poderia pagar menos do que aquele que utiliza mais. Mesmo que a nuvem tenha que reservar uma certa quantidade de recurso para um cliente, enquanto o mesmo não utiliza do seu percentual concedido, a nuvem lucra com a economia de energia realizada pelo cliente, mantém com maior facilidade os acordos de SLA e além de que os recursos estão sub-utilizados. Com o estudo do modelo de tarifação da *Amazon EC2* e dos trabalhos acadêmicos, que serão apresentados na secção 2.2), chegou-se a conclusão que a *Amazon EC2* e as demais provedoras de nuvens poderiam utilizar um modelo de tarifação mais flexível, o que poderia aumentar a satisfação dos clientes, por meio de uma melhor relação entre o consumidor e o provedor. Na próxima seção serão apresentados os estudos realizados pela academia a respeito da visão de modelo de tarifação mais adequada para computação em nuvem.

2.2 Trabalhos acadêmicos

A tarifação em nuvem ainda é um campo recente de pesquisa. Porém, durante a fase inicial deste estudo foram realizadas pesquisas nas quais foram encontrados artigos e trabalhos que já exploraram o tema. Nesta subseção são apresentados alguns destes trabalhos que contribuem para a proposta *aCCountS*.

O trabalho de Silva *et al.* (SILVA et al., 2012) teve um papel importante neste levantamento inicial ao apresentar um estudo de mapeamento sobre a tarifação em computação em nuvem. Neste mapeamento foram encontrados 23 artigos, desses, os mais importantes para esse trabalho são os artigos de Ruiz-Agundez *et al* (I.R.; Y.K.; P.G., 2006), de Elmroth *et al* (ELMROTH et al., 2009) e de Caracas e Altmann (CARACAS; ALTMANN, 2007). Além desses, contribuíram nesta proposta os trabalhos de Yu e Bhatti (YU; BHATTI, 2010), Narayan (NARAYAN et al., 2012) e do grupo de pesquisa e desenvolvimento de software *Hartmann Software Group* (GROUP, 2012).

A proposta de Ruiz-Agundez *et al.* consiste em um processo de tarifação para recursos computacionais. Este trabalho é apontado como a única referência para uma taxonomia completa de um processo de contabilização e tarifação de recursos, conforme descrito na Figura 2.3.

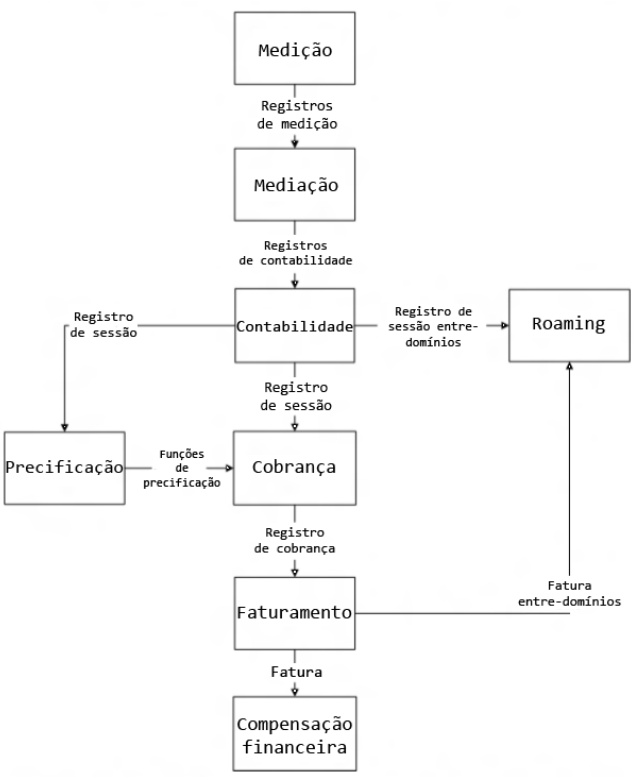

Figura 2.3: Fluxo de tarifação, adaptado de Ruiz-Agundez *et al* (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011)

O fluxo de tarifação proposto por Ruiz-Agundez *et al* (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011) é composto de oito tarefas, nas quais o resultado de uma serve como entrada para uma tarefa seguinte. A primeira tarefa é a *medição* e sua função é monitorar o uso de recursos na nuvem. Estes recursos podem ser o consumo de CPU ou memória de uma determinada máquina virtual. A segunda tarefa é a *mediação*, que refina os dados recebidos da tarefa anterior, transformando-os em dados mais fáceis de serem manipulados pelas próximas etapas. Após a mediação, a tarefa de *contabilidade* tem por funções a filtragem, a coleta e a agregação das informações sobre o uso de recursos por um determinado cliente. Como resultado de sua execução, registros de sessão são enviados para a *cobrança*, que realiza as tarefas relacionadas a geração dos registros de cobrança para um cliente específico. Caso esses registros calculados sejam de um cliente de outra máquina virtual ou nuvem (consumo de recursos de outras máquinas para alcançar a escalabilidade), os dados são repassados para a tarefa *roaming*, cuja função é enviá-los para a máquina virtual ou nuvem onde o cliente possua cadastro e realize o cálculo da cobrança específico. A tarefa de *precificação* é responsável por definir as operações que serão realizadas no cálculo da tarifação. A tarefa *cobrança* realiza o cálculo da cobrança a partir dos dados medidos pela nuvem (contabilidade) e dos valores dos recursos monitorados (precificação). Em seguida, tarefa de *faturamento* recebe os registros de cobrança realizados durante um período de tempo, para então calcular a fatura completa. Por fim, a fatura também é enviada à tarefa de *compensação financeira*, que realiza o cumprimento do pagamento da fatura.

Ruiz-Agundez *et al* (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011), também sali-

enta a importância de tornar a tarifação flexível, levando em consideração diferentes requisitos de nuvem como: a contabilização pelo uso dos recursos, incentivos e descontos para usuários e a necessidade de permitir alterações das políticas de forma fácil e automatizada. Esses requisitos foram assimilados e serviram de base para a construção da linguagem e do serviço de tarifação propostos.

Outra questão importante tratada pelos autores é a forma como a política de tarifação é implementada. Segundo Ruiz-Agundez *et al*, cada provedor de nuvem desenvolve seu sistema de contabilização próprio e o codifica diretamente na infraestrutura da nuvem, dificultando manutenções na política de negócios. Entretanto, para validar seu modelo, o fluxo descrito na Figura 2.3 foi implementado em um serviço de tarifação comercial, chamado de *JBilling* (JBILLING, 2013), o que impede que maiores detalhes a respeito da solução possam ser estudados. No mais, sabe-se que o *JBilling* é baseado em sistemas de regras de negócios - BRMS (do inglês, *Business Rule Management System*) e não é o mais indicado para definições de regras mais complexas de tarifação, como será explicado posteriormente nessa seção. Esses fatores, de certa forma, motivaram a criação da DSL e do serviço de tarifação propostos, visando possibilitar a criação e alteração nas políticas de negócio, sem modificar o código da aplicação, apenas alterando as políticas de tarifação definidas no serviço por meio da DSL.

A proposta de Elmroth *et al.* (ELMROTH et al., 2009), um modelo arquitetural de faturamento do RESERVOIR (*Resources and Services Virtualization without Barriers*), é um projeto de pesquisa com foco na federação de nuvens no nível de infraestrutura (ELMROTH et al., 2009), e que tem como objetivo propor um modelo arquitetural para contabilização e faturamento em federação de nuvens.

Esse modelo é composto por três partes. O primeiro, *Service Manager* - “Gerente de serviço”, é camada responsável por implantar as aplicações, gerenciá-las, monitorar o consumo de recursos, verificar e aplicar o SLA. A segunda, o *Virtual Execution Environment Manager* - "Gerente do ambiente de execução virtual", é o responsável por otimizar e gerenciar a localização dos “ambientes de execução virtual” (do inglês, *Virtual Execution Environment*), tanto na máquina local como em máquinas remotas. O terceiro, chamado de *Virtual Execution Environment Host* - "Hospedeiro do ambiente de execução virtual", é a camada responsável por executar e monitorar o ambiente de execução virtual, repassando os dados de monitoramento para a camada acima e gerenciando esse ambiente.

Baseado no SGAS, sistema de contabilidade em grade, (do inglês, *Scalable Grid-wide capacity allocation with the SweGrid Accounting System*), o autor cria um modelo de faturamento para uma federação de nuvens em cima da arquitetura proposta para o *RESEVOIR*. Esse modelo arquitetural (Figura 2.4) possui três componentes: *(i)* o primeiro, a Camada de Contabilidade, é responsável por coletar os dados de recursos usados e dados sobre SLA; *(ii)* o segundo, a Camada de Faturamento, possui quatro componentes (engenharia de pós-pago, engenharia de pré-pago e mais dois que também fazem parte da camada de negócios (*Service Configuration Analyzer* - Serviço de análise de configuração) e (*Service Life-cycle Manager* - Serviço de Gerenciamento de Ciclo de Vida) e *(iii)* o terceiro; a Camada de negócio, é tanto responsável pela parte de análise do SLA, quanto da interface com o usuário, fornecendo um ambiente para implantação de serviços, escolha do modelo de pagamento, compra de créditos,

monitoramento do sistema e o uso de recursos.

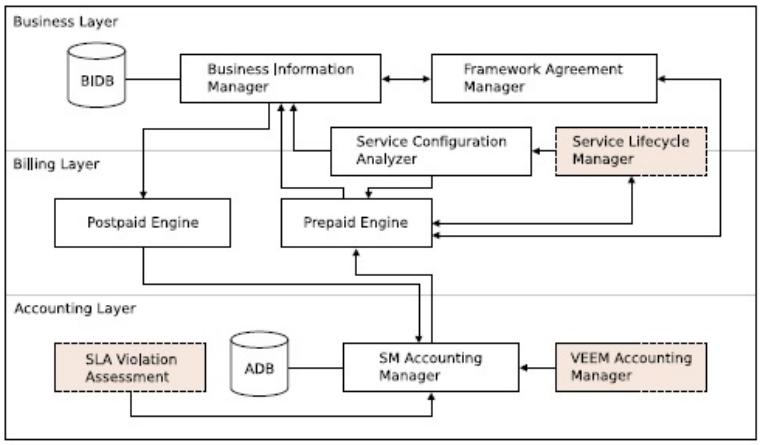

Figura 2.4: Modelo arquitetural de faturamento para o *RESERVOIR* (ELMROTH et al., 2009).

Dois tipos diferentes de modelo de pagamento foram propostos para o projeto *RESERVOIR*: pagamento pré-pago, em que o usuário compra créditos que vão sendo decrementados de acordo com a utilização dos recursos do sistema, e pagamento pós-pago, onde os recursos consumidos pelo usuário vão sendo contabilizados para, no fim de um período, realizar o faturamento. Essa, portanto, é uma contribuição do *RESERVOIR* para este trabalho, onde os modelos de pagamento pré-pago e pós-pago poderão ser utilizados na definição de uma política de tarifação. Na proposta, quando o modelo de pagamento é pré-pago, a camada de faturamento verifica antes da execução do serviço, se o crédito do cliente é suficiente para realizar a tarefa. Se a condição for verdadeira, a atividade é realizada e o valor do serviço é debitado do crédito do cliente. Caso contrário, o serviço não é executado. Entretanto, não fica claro no artigo como o sistema sabe a quantidade de recursos que serão necessários para uma determinado trabalho e o custo do mesmo para verificação de crédito no modelo pré-pago.

Elmroth et al. (ELMROTH et al., 2009) identificaram alguns requisitos funcionais que devem ser atendidos na construção de um modelo de faturamento para uma federação de nuvens. Entre eles, alguns contribuíram para a criação da linguagem e do serviço de tarifação propostos, são estes: *(i)* o modelo de faturamento deve ser aberto para manutenções futuras; *(ii)* o formato de dados utilizado no modelo de faturamento deve ser de tal forma que tanto os dados de *hardware* (exemplo: consumo de *cpu* e memória), como os dados de performance *KPI* (exemplo: transações no banco de dados por segundo) sejam manuseados pelas funções desse processo, *(iii)* o modelo de faturamento deve contabilizar tanto os recursos usados como as compensações por quebra de acordo SLA e *(iv)* o faturamento de um serviço deve ser contabilizado com base no serviço e no modelo de pagamento.

A proposta de Yu e Bhatti (YU; BHATTI, 2010) busca medir o consumo de energia gasta por um cliente (seu consumo) e deixá-lo à par dessa informação, tanto a título de conhecimento, como com a finalidade de descontos no pagamento da fatura. A provedora de nuvem pode optar por serviços que consumam menos energia ou incentivar o cliente a gerenciar a utilização dos recursos com essa finalidade. A questão do incentivo à economia de energia, apontada por Yu e Bhatti foi assimilada na definição das políticas de tarifação dessa proposta,

que estimula a configuração de máquinas virtuais, de forma a economizarem mais energia, segundo algumas técnicas apontadas no trabalho de Imada *et. al.* (IMADA; SATO; KIMURA, 2009).

O trabalho de Narayan *et al.* (NARAYAN et al., 2012) propõe um modelo de tarifação em que a cobrança dos serviços é baseada na carga de utilização dos recursos (cpu, memória, armazenamento, rede, entre outros) pelo cliente em função do tempo e os preços são proporcionais a quantidade consumida do recurso na infraestrutura de nuvem. O autor propõe um componente de tarifação para a nuvem, baseado na proposta de cobrança, definindo a tarifação a partir de algoritmos de precificação, como definida no algoritmo representado na Figura 2.5(a) e cobrança, como ilustrado na Figura 2.5(b). Esses algoritmos são implementados diretamente na infraestrutura da nuvem, necessitando reimplementação para adição de novas formas de precificação ou atualização das existentes, tornando a tarifação inflexível e trabalhosa.

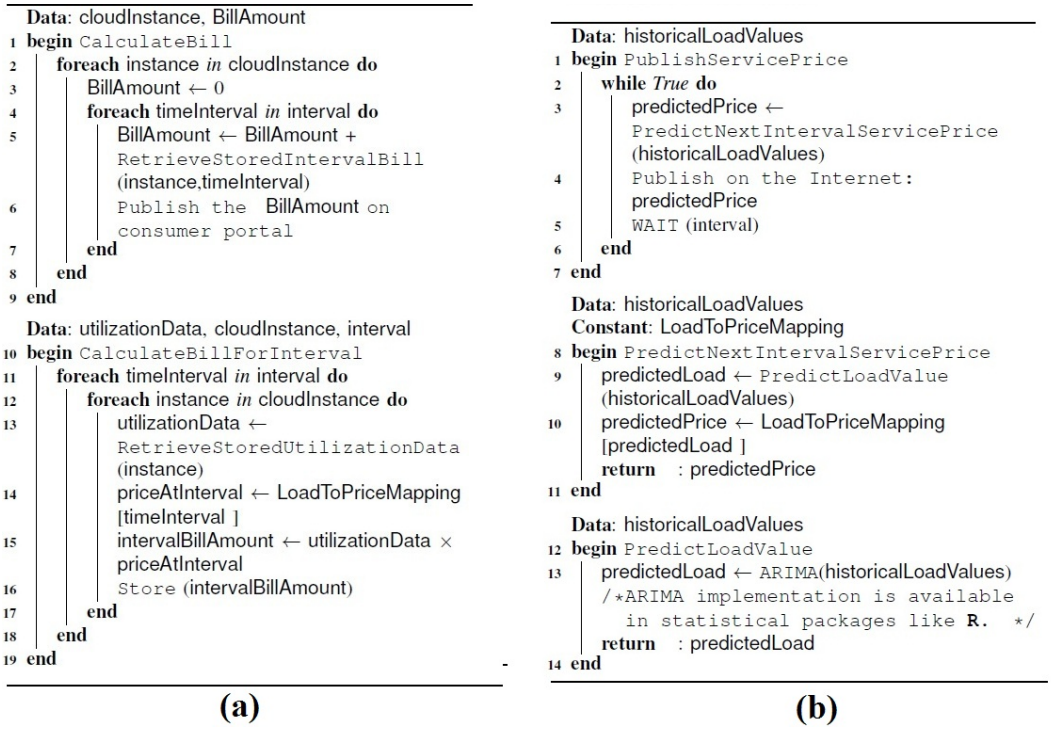

Figura 2.5: Algoritmos de (a) precificação e (b) cobrança para tarifação em nuvem (NARAYAN et al., 2012)

Assim sua proposta se parece com as expectativas dos demais estudiosos, em que a precificação e a cobrança devem levar em conta o uso dos recursos consumidos. Tendo, portanto, contribuído com os requisitos de tarifação propostas nesta dissertação.

Caracas e Altmann (CARACAS; ALTMANN, 2007) propõem um serviço de precificação para serviços de computação em grade. Nele foi descrito os requisitos funcionais, a arquitetura, e as interfaces do serviço de precificação. O serviço de precificação permite expressar o esquema geral de preços proposto como um documento XML, que pode ser ligado a acordos de nível de serviço. Ao contrário de outras propostas sobre tarifação, o serviço de precificação é separado da funcionalidade de medição, contabilidade e pagamento. E para validar o conceito proposto foram realizados dois cenários, o primeiro constitui no uso da grade por

meio da computação utilitária e o segundo, validação da utilização dos recursos por meio do modelo de tarifação proposta.

Segundo Caracas e Altmann, uma política de tarifação de uma provedora de nuvem busca alcançar tipos diferentes de finalidades, tais como maximizar os lucros, maximizar o bem estar social ou definir um esquema de preço justo. Nesse artigo foi proposto um modelo de tarifação para serviços em grade através de uma quádrupla (Q,T,C,U). Nela, o Q significa quantidade de recursos consumidos, representados pela tupla QL e QM, onde QL representa um limite para a quantidade de recursos a ser consumido e QM representa a quantidade consumida acima de QL. T, tempo de consumo dos recursos e é representado por um vetor de tempos: tc, ts e td, onde tc é o tempo atual, ts é a hora de início e td é o tempo de duração. C, classe da qualidade do serviço e U, perfil do usuário. Através de uma precificação dinâmica, os recursos podem ser mais ou menos importantes, dependendo da quantidade de requisições solicitadas em um determinado momento.

Na Figura 2.6 é mostrado o processo de tarifação de serviços proposto por Caracas e Altmann para computação em grade. O processo inicia-se pela etapa 0, onde o administrador do prestador de serviços publica os esquemas de preços e SLAs. Para utilizar o serviço, passo 3, o cliente terá que: 1, obter uma negociação de conta e 2, obter um SLA. Quando o cliente quer obter um SLA para um determinado recurso, o serviço de SLA consulta o serviço de precificação para obter uma lista de esquemas de preços. O cliente assina um SLA para a respectiva quantidade de recursos. Os valores dos preços nos regimes de preços são definidos pelo serviço de precificação do lado do provedor. O serviço de informação de preços separa a funcionalidade de fixação de preços e definição de regimes de preços do processo de definição e assinatura de SLAs. Este recurso permite ajustar dinamicamente os preços para o perfil de harware e o perfil da aplicação (mais simples ou mais robusta) com base nos preços de mercado e em outros fatores em ambientes de negócios com rápidas mudanças, mantendo o SLA existente.

O trabalho de Caracas e Altmann, portanto, trata-se de um sistema de difícil manutenção, em que o administrador precisa codificar as regras de negócio diretamente na infraestrutura do *middleware* para grade utilizado na implementação, não sendo simples a definição e alteração de políticas de tarifação. Contudo as ideias de cobrança utilizadas, como a quádrupla (Q,T,C,U) e as questões de SLA contribuíram para a definição da linguagem e serviço propostos neste trabalho.

2.3 Discussão

Para analisar as diferentes proposta de tarifação da academia e da indústria e compará-las com a deste trabalho, na Tabela 2.1 são apontados os diferentes requisitos de tarifação, que através dos estudos realizados, foram considerados relevantes para tarifação em nuvem e foram selecionados os trabalhos que atendem tais requisitos.

Os requisitos apontados como importantes para tarifação em nuvem são *(I)* Tarifação por tempo e uso individual dos recursos, *(II)* Fácil alteração das políticas de negócio, *(III)* Modelo de pagamento pré-pago e pós-pago, *(IV)* SLA na tarifação, *(V)* Bem estar Social,

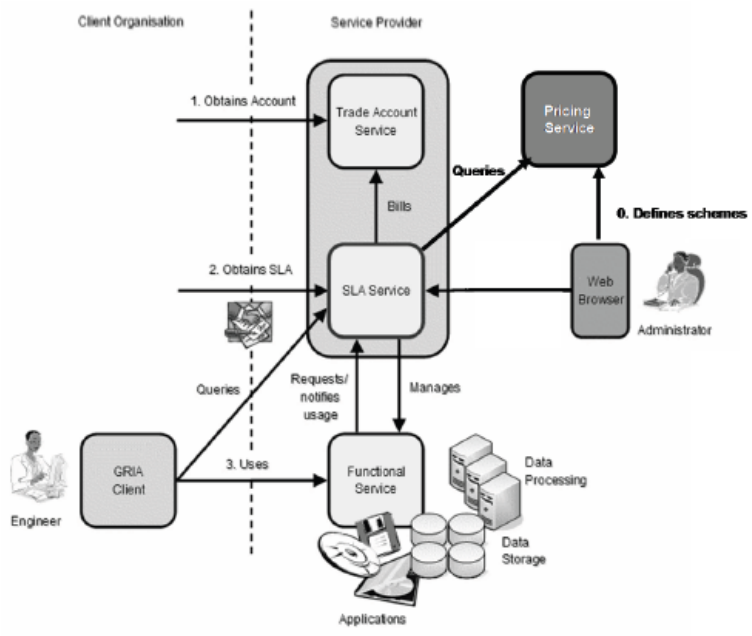

Figura 2.6: Serviço de precificação proposto por Caracas e Altmann(CARACAS; ALTMANN, 2007)

(VI) Tarifação de software, serviços e recursos, *(VII)* Serviço que suporta regras de tarifação complexas, *(VIII)* Processamento dinâmico das políticas.

Tabela 2.1: Comparação das propostas de tarifação existentes

Propostas ou soluções	*I*	*II*	*III*	*IV*	*V*	*VI*	*VII*	*VIII*
Ruiz-Agundez (I.R.; Y.K.; P.G., 2006)	■		■					
Elmroth et al. (ELMROTH et al., 2009)	■		■	■		■		
Yu e Bhatti (**??**)					■			
Narayan (NARAYAN et al., 2012)	■							
Caracas e Altmann (CARACAS; ALTMANN, 2007)	■			■	■			
Serviços baseados em BRMS (GROUP, 2012)	■	■	■	■	■	■		
Serviço de tarifação da *Amazon EC2* (AMAZON, 2013)						■	■	■

Visto todos esses trabalhos, verificou-se que existem propostas que buscam tratar diferentes requisitos de forma individual para a tarifação em computação em nuvem, seja os modelos encontrados na indústria quanto as propostas da academia. Com isso, vislumbrou-se nesta dissertação a realização de um serviço que contemple todas esses requisitos em uma única solução, que será apresentada no Capítulo 3.

2.4 Conclusão

Dado o modelo de tarifação na *Amazon*, que monitora, para fins de recolhimento somente o tempo de utilização dos recursos de *harware*, além disso, não leva em consideração o

uso individual de cada recurso e não atende diversos outros requisitos de tarifação considerados pela academia,

Por meio do estudo dos modelos de tarifação utilizados na indústria percebeu-se que um novo modelo de tarifação, que seja mais flexível, apresentando novos requisitos de cobrança seria importante, tais como tarifação por uso de cada recurso de *hardware* ou a consideração de questões sobre a economia de energia, apontados pela academia, são relevantes para a cobrança pelo uso dos serviços. Estes requisitos poderiam ser utilizados na indústria com a finalidade de atender diferentes perfis de clientes, acatar políticas distintas de negócio da provedora de nuvem e incentivar tanto o uso da nuvem, como questões sociais interessantes apontados pelos artigos científicos.

Entretanto, outras características do modelo de cobrança da indústria foram vistas como consideráveis para se tarifar serviços de infraestrutura, tais como: disponibilizar diferentes perfis de máquina para atender necessidades distintas dos cliente ou tarifar os recursos dependendo da garantia e fidelidade dos mesmos à nuvem, ou vice-versa, como as instâncias sob-demanda, reservada e *spot*.

Baseado nisso, percebeu-se que tanto a indústria como a academia trazem requisitos importantes na tarifação em nuvem, sendo relevante considerar ambas as propostas, oferecendo um modelo flexível em que a provedora de nuvem pudesse prover diferentes formas de cobrança a fim de atender as distintas políticas de negócio e alcançar vários perfis de clientes.

Então, este trabalho propõe um modelo de tarifação flexível para permitir a criação de diferentes políticas de cobrança pelas provedoras de nuvem. Isso a partir de um serviço de tarifação e uma DSL de tarifação em nuvem utilizada pelo mesmo. No Capítulo 3 será abordada a proposta deste trabalho de mestrado.

3 ACCOUNTS

Neste capítulo será explicada a proposta deste trabalho de mestrado, o *aCCountS*. Ele foi projetado com o objetivo de flexibilizar a tarifação para nuvens de IaaS, por meio da definição de políticas, usando o serviço de tarifação implementado (*aCCountS-Service*), que utiliza a *aCCountS-DSL*. Para isso, foi necessário observar como a cobrança de serviços de IaaS é realizada na indústria e o que a academia já havia pesquisado sobre o assunto e quais seriam os principais problemas dos trabalhos propostos, bem como indicativos de possíveis soluções para tais problemas.

Com essas pesquisas, apresentadas no Capítulo 2, entendeu-se que uma provedora de nuvem deve ter a liberdade para tarifar seus clientes de acordo com suas políticas de negócio, mas a forma de cobrança pelo uso dos serviços devem ficar claras para o cliente, que poderá escolher entre as diferentes provedoras, àquela na qual as políticas de cobrança e os serviços satisfazem seus interesses, tantos os financeiros como os de necessidade computacional.

Percebeu-se que a tarifação em nuvem por parte da indústria está consolidada, isto é, seguindo as diretrizes de sua maior representante, a *Amazon EC2*. Entretanto, na visão da academia, a mesma é inflexível (CARACAS; ALTMANN, 2007), quando, por exemplo, a cobrança realizada nessas provedoras não disponibilizam tarifação pela quantidade de recursos de hardware (CPU, memória, armazenamento, entre outros) consumidos. Por exemplo, dois clientes que requisitam instâncias iguais na nuvem pagam o mesmo preço, mesmo o primeiro utilizando 90% dos recursos da máquina virtual instanciada, enquanto o segundo usa apenas 10% de uma instância semelhante (AMAZON, 2013).

Outras questões se referem à preservação do meio ambiente, a satisfação dos clientes e o bem-estar social (preocupação em atender as necessidades do cliente e do meio ambiente). Por exemplo: quais empresas no mercado estão incentivando a economia de energia, que pode ser conseguida com configurações na criação das máquinas virtuais, por exemplo (IMADA; SATO; KIMURA, 2009). Os clientes mais assíduos ou com maior consumo estão sendo valorizados? As questões de SLA estão sendo medidas e entendidas pelo cliente?

Estas questões, e outras mais, ainda não estão bem esclarecidas na computação em nuvem. Como resultado, cada provedora de serviços de nuvem acaba por ter que implementar sua solução de tarifação. Baseado nisso, propõe-se uma maneira para que a provedora de nuvem crie sua forma de cobrança de acordo com seus interesses. A proposta visa atender a um nível de flexibilidade que atenda a diferentes objetivos, como o *(i)* lucro imediato, *(ii)* questões considerando o bem estar social, preservação do meio ambiente e o respeito aos clientes, ou ainda, *(iii)* cobranças mais justas, considerando todos os recursos consumidos pelo cliente. Além disso, deve ser simples e de fácil utilização, diferente dos serviços baseados em BRMS, que além de serem genéricos, possuem uma curva de aprendizado não suave.

Para isso criou-se *(i)* uma DSL de tarifação, com qual podem ser criadas políticas para atender diferentes objetivos e *(ii)* um serviço de tarifação que dá suporte a esta linguagem. O serviço de tarifação em nuvem, possui seu fluxo de execução na proposta de Ruiz-Agundez *et al.*, como visto no Capítulo 2. O serviço é formado por dois componentes, o agente de

monitoramento e o componente de tarifação. O agente monitora os recursos das máquinas virtuais na infraestrutura de nuvem e o componente interpreta regras de tarifação definidas pela provedora do serviço (administrador), contabiliza a utilização destes recursos e gera uma conta de tarifação para o cliente. O segundo elemento é a linguagem de domínio específico, chamada ***aCCountS-DSL***, que permite a administradores de nuvem definirem regras de tarifação para recursos de sua infraestrutura. A Figura 3.1 mostra uma visão geral sobre o *aCCountS*.

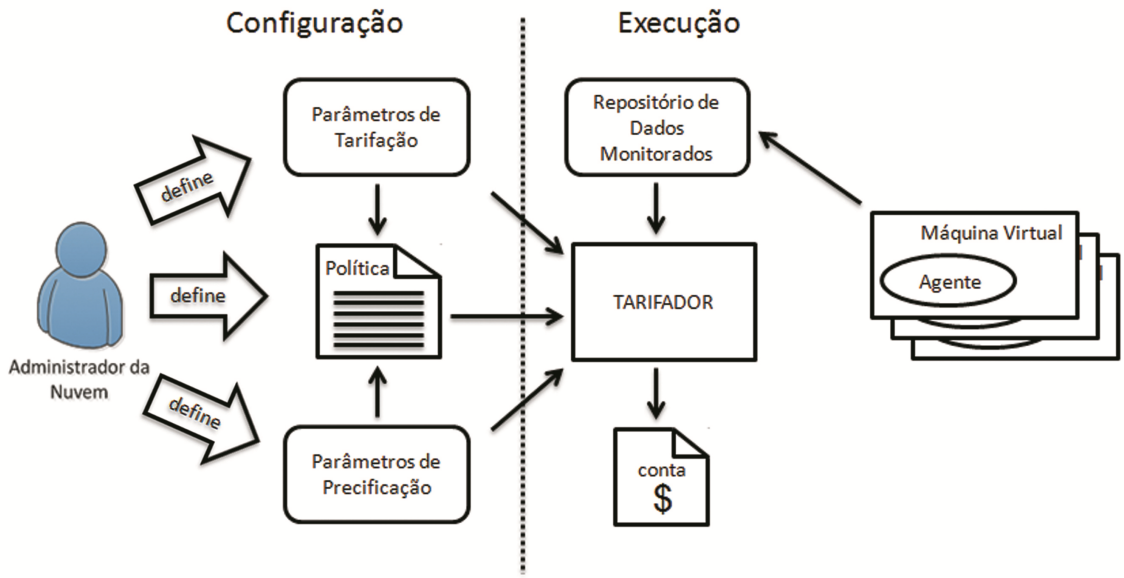

Figura 3.1: Visão Geral da Proposta.

Em linhas gerais, pode-se separar a utilização do serviço em duas etapas. Na etapa inicial é feita sua configuração. Nela, o administrador da nuvem precisa definir os parâmetros de tarifação, ou seja, quais serão os recursos monitorados nas máquinas virtuais de sua nuvem, e que serão utilizados para composição das regras de tarifação. Em geral, estes recursos incluem uso de CPU, memória, disco, tempo de uso, transações em bancos de dados, serviços e software utilizados, dentre outros.

Após esta definição, faz-se necessário estabelecer valores (parâmetros de precificação) para os recursos que serão monitorados. Na proposta do *aCCountS*, a precificação é fortemente atrelada ao conceito de perfil de uma máquina virtual. Este perfil refere-se às possíveis configurações das máquinas virtuais que podem ser instanciadas na nuvem, de forma semelhante ao que acontece em infraestruturas de nuvem conhecidas, como *Amazon EC2* ou *GoGrid*. Por exemplo, na *Amazon AWS*, o usuário pode escolher entre instâncias do tipo Micro, Pequena, Média, Grande e ExtraGrande(AMAZON, 2013). Cada uma destas instâncias tem valores diferentes para seu uso.

Com estes parâmetros definidos, o administrador pode criar as políticas de tarifação de sua nuvem utilizando a *aCCountS-DSL*. Neste caso, as políticas definem as diferentes práticas em relação a cobrança pelo uso de recursos de suas máquinas virtuais instanciadas. Estas políticas podem incluir uma tarifação apenas pelo tempo de uso de uma máquina virtual, ou podem agregar taxas em relação ao uso dos recursos de rede, média de consumo de memória, quantidade de transações no banco de dados, além de descontos por economia de energia,

quebra de regras de SLAs ou perfil do cliente. Almeja-se utilizar os diferentes requisitos de contabilização para criar uma política flexível.

A segunda etapa diz respeito à execução do serviço. Para isso, em cada máquina virtual, um agente é responsável por coletar informações dos parâmetros de tarifação e repassá-las a um repositório. De posse das informações sobre consumo de recursos das máquinas virtuais e de como os recursos devem ser tarifados, o tarifador pode então gerar uma conta de consumo para clientes da nuvem. Com isso, primeiramente precisa-se realizar a configuração e então, a execução pode ser iniciada. No entanto, o serviço pode executar normalmente enquanto o administrador atualiza os dados configurados ou cadastra novos parâmetros, políticas e preços.

O fluxo de contabilização de Ruiz-Agundez *et al.* (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011) foi adaptado para atender à proposta do *aCCountS*. Com isto, o fluxo de contabilização deste trabalho está ilustrado na Figura 3.2.

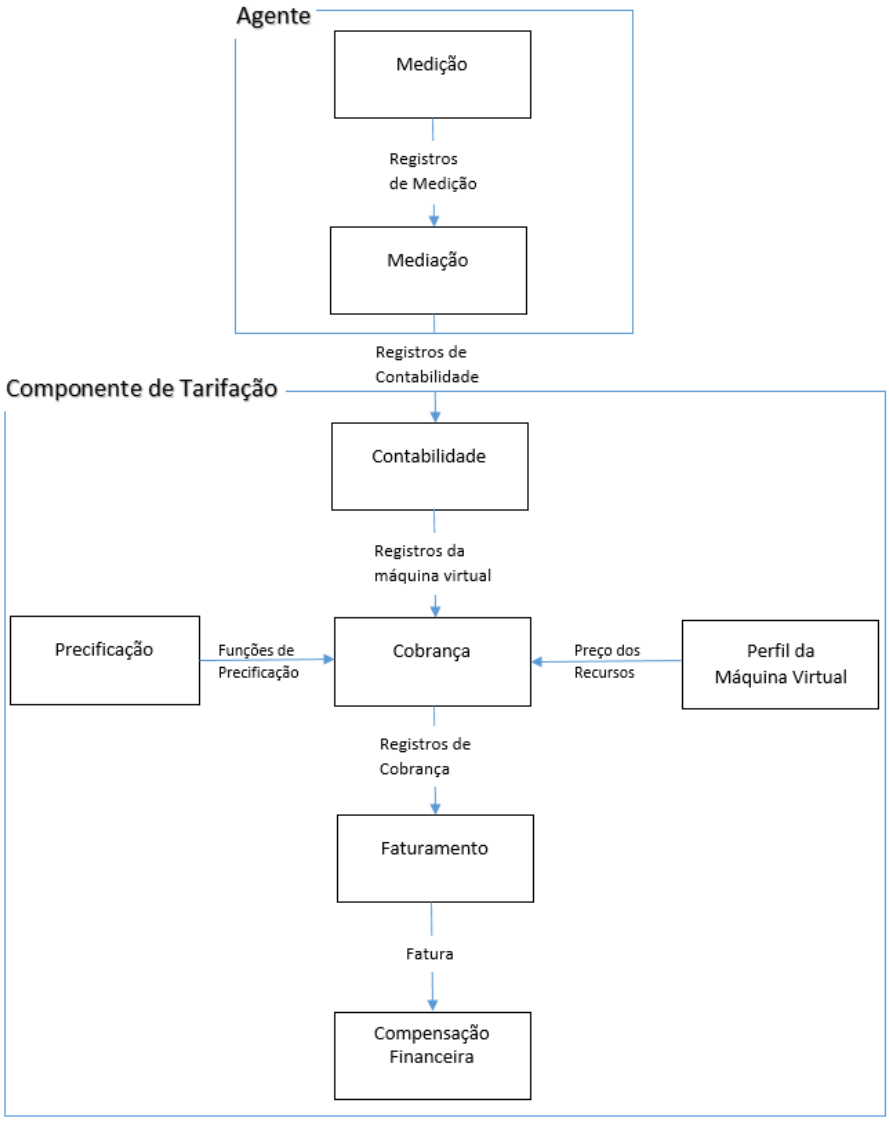

Figura 3.2: Fluxo do *aCCountS* baseado no fluxo proposto por Ruiz-Agundez (RUIZ-AGUNDEZ; PENYA; BRINGAS, 2011).

Esse fluxo foi definido para serviços de contabilização de nuvens, no qual o agente e o componente de tarifação estão em ambientes físicos diferentes e a contabilização é realizada para cada máquina virtual instanciada. Com isto, algumas modificações foram realizadas, estão citadas a seguir:

- *função de roaming*: é dispensada, pois qualquer uso de recursos na nuvem por meio das máquinas virtuais é reportado para o mesmo lugar, o *aCCountS-Service*;
- *função de precificação*: diferente do modelo de Ruiz Agundez *et al.*, ela não recebe os registros de contabilização, apenas gera a função de precificação utilizada pela função *cobrança*;
- *função de perfil*: adicionada ao modelo com a finalidade de informar à função *cobrança* os preços dos recursos para o cálculo da conta;
- *funções separadas fisicamente*: devido ao processo ser realizado em ambientes físicos diferentes (agente e componente de tarifação), algumas funcionalidades precisam estar no lado do agente (na nuvem) e outras no lado do componente de tarifação. O agente abrange as funções de medição e mediação, pois a primeira precisa estar funcionando na nuvem para monitorá-la e a segunda, por questões de otimização, necessita ficar próxima da primeira, evitando grande quantidade de transferência de dados através da rede. As demais funcionalidades pertencem ao componente de tarifação que recebe os recursos medidos pela máquina virtual do cliente e os tarifa de forma a gerar uma fatura a ser paga.

Para ilustrar o fluxo que ocorre no agente, a Figura 3.3 detalha as entradas e saídas e o processamento interno para o agente.

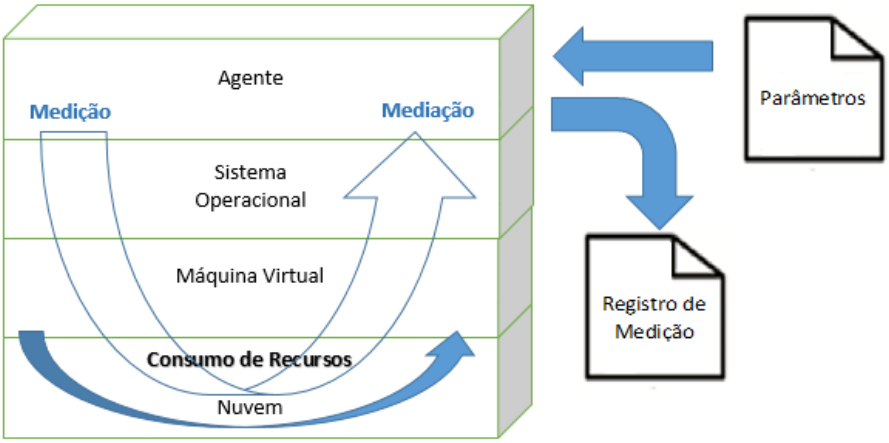

Figura 3.3: Fluxo de tarifação no agente.

A nuvem possui várias máquinas virtuais e cada uma delas possui seu sistema operacional hospedeiro. O agente deve ser instalado no sistema operacional da máquina virtual e por meio dele, realizar as funções de medição e mediação. A função de medição recebe os

parâmetros, ou seja, os recursos que devem ser medidos e como medi-los. A partir dessas informações, o agente realiza a aferição do consumo de recursos (*hardware*, *software*, serviços) da máquina virtual implantada na nuvem. A função de mediação transforma esses dados em um formato mais fácil de ser trabalhado e os envia para o componente de tarifação.

Sabe-se que o agente precisará consumir recursos da VM para realizar suas funcionalidades. Para atenuar este impacto, a periodicidade do monitoramento é configurável. A relação entre a frequência de execução do monitoramento e a granularidade dos dados obtidos consiste em um *tradeoff*. Quanto menor a frequência, melhor será o controle sobre picos de uso. Porém, maior será o impacto sobre o desempenho da VM.

O componente de tarifação, como já foi explicado, é gerenciado pelo administrador da nuvem, que configura: *(i)* os parâmetros que devem ser monitorados nas máquinas virtuais, *(ii)* as políticas de tarifação e *(iii)* os preços dos recursos disponibilizados na nuvem. Várias combinações de políticas e perfis podem ser criadas pela provedora de nuvem e ofertadas para atender às diferentes necessidades dos clientes.

Com isso, o *aCCountS-Service* pode realizar seu fluxo de cobrança. O componente de tarifação, então, recebe os registros de medição do agente e os envia à função de cobrança, que por sua vez, utiliza a função de precificação para calcular a cobrança por meio desses registros e dos preços dos recursos. Com os registros de cobrança, a função de faturamento calcula a fatura do cliente, a qual é a saída do serviço de tarifação. A função de precificação especificada é gerada a partir da política de tarifação definida por meio da linguagem *aCCountS-DSL*.

Para melhor especificar cada uma destas etapas, as próximas subseções abordarão em mais detalhes *(i)* a arquitetura do serviço proposto e *(ii)* a linguagem de domínio específico, *aCCountS-DSL*.

3.1 Visão arquitetural do *aCCountS*

A arquitetura do *aCCountS* é representada na Figura 3.4, por meio de um diagrama de componentes UML. Nesta figura, os elementos marcados com o estereótipo *Entity* representam abstrações que denotam dados manipulados na proposta *aCCountS*. Os demais componentes da arquitetura assumem o papel de *gerenciadores* destes recursos e exercem papel de processamento das etapas do processo de tarifação. A arquitetura *aCCountS* possui dois macro-componentes: *(i)* o *aCCountS-Agent* e o *(ii) aCCountS-Service*. O primeiro representa o agente de coleta de informações, e que deve estar presente em cada máquina virtual em execução na nuvem. O segundo representa um serviço que recebe os dados monitorados, e que através da configuração de seus sub-componentes, consegue gerar a tarifação das máquinas virtuais associadas a um determinado cliente.

A divisão do *aCCountS* em macro-componentes foi motivada pela ideia de um serviço de tarifação em nuvem projetado de forma que o serviço seja independente de uma infraestrutura de nuvem específica. Com isto, objetiva-se a concepção de um serviço reutilizável, que pode ser utilizado para realizar o processo de tarifação para diferentes nuvens de infraestrutura.

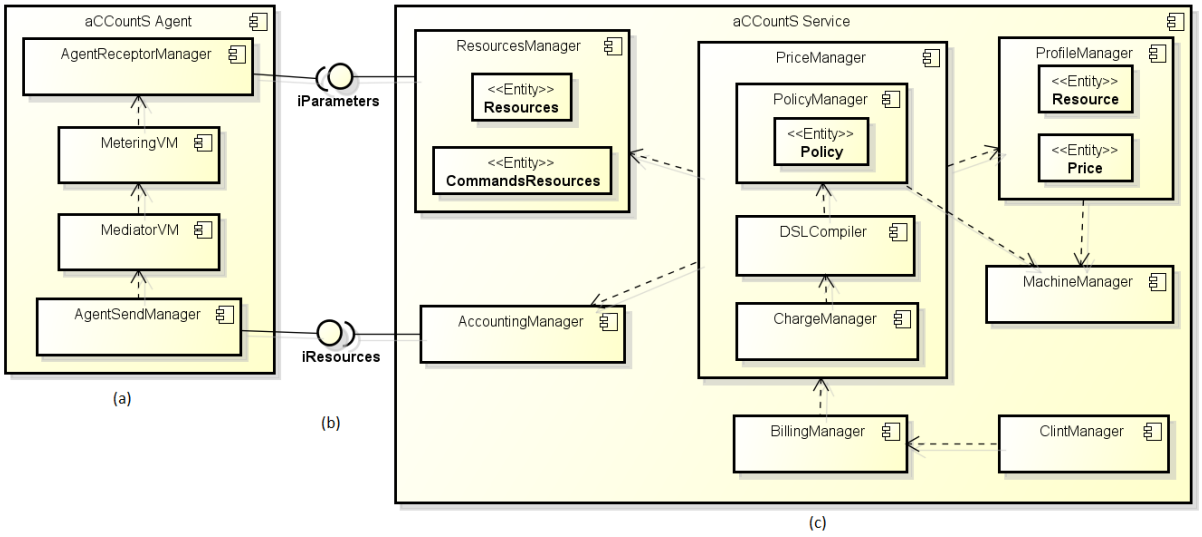

Figura 3.4: Diagrama de componentes UML com os principais elementos de *aCCountS*.

Esse serviço visa atender a diferentes necessidades, como:

1. **Provedoras recém-criadas**: Provedoras de IaaS recém-criadas podem utilizar o *aCCountS* para realizar a tarifação de seus serviços, contando com os benefícios de uma ferramenta de tarifação já construída;

2. **Provedoras com dificuldades na atualização de políticas de negócios**: A maioria das empresas de nuvem, segundo Ruiz-Agundez (I.R.; Y.K.; P.G., 2006), implementa suas políticas diretamente na infraestrutura de nuvem, precisando de uma reprogramação quando necessário modificar essas políticas. O *aCCountS* permite à provedora criar novas políticas, alterar e excluir as já existentes, sem alterações no código do sistema;

3. **Provedoras que utilizam serviços de tarifação baseadas em BRMS e não estão satisfeitas com as limitações do serviço**: Os serviços de tarifação baseados em BRMS têm suas vantagens e podem ser uma boa solução para algumas empresas. Entretanto, esse tipo de sistema é a complexo, além disso sua curva de aprendizado é longa para que usuários consigam dominar a definição de suas regras de negócio. A definição de uma linguagem específica para tarifação, com recursos voltados exclusivamente para definição das regras de cobrança deve facilitar o entendimento e criação das políticas de tarifação. A *aCCountS-DSL* permite basicamente a definição de operações matemáticas e lógicas sobre recursos e valores definidos de preços para tais recursos, facilitando sua utilização.

Para utilizar o serviço do *aCCountS-Service*, a provedora precisa de um agente para monitorar sua nuvem e repassar os dados de consumo para o serviço. Esse agente pode ser implementado pela própria provedora de nuvem ou utilizar o *aCCountS-Agent*, fornecido como parte da solução de tarifação. Com isso, a contabilização de cada máquina virtual ocorre de forma paralela, sejam quantas nuvens e quantas máquinas virtuais forem instanciadas. De modo a facilitar seu monitoramento, cada máquina virtual é identificada de forma independente e única pelo serviço, por meio de um identificador gerado no *aCCountS-Service* para

realizar a associação entre uma máquina virtual e, sua política de tarifação e perfil de *hardware*. (O administrador é responsável por criar as políticas de tarifação para nuvens, seus perfis de máquina e as configurações de tarifação que serão associadas às máquinas virtuais. A configuração de uma máquina ocorre quando o administrador combina um perfil e uma política para contabilizar uma VM com as mesmas características de *hardware* do perfil concebido. Nessa VM será instalado o agente configurado com o identificador gerado, que associa a máquina à configuração de tarifação definida). A Figura 3.5 ilustra o diagrama de atividades do processo de tarifação do *aCCountS* para cada máquina virtual independente.

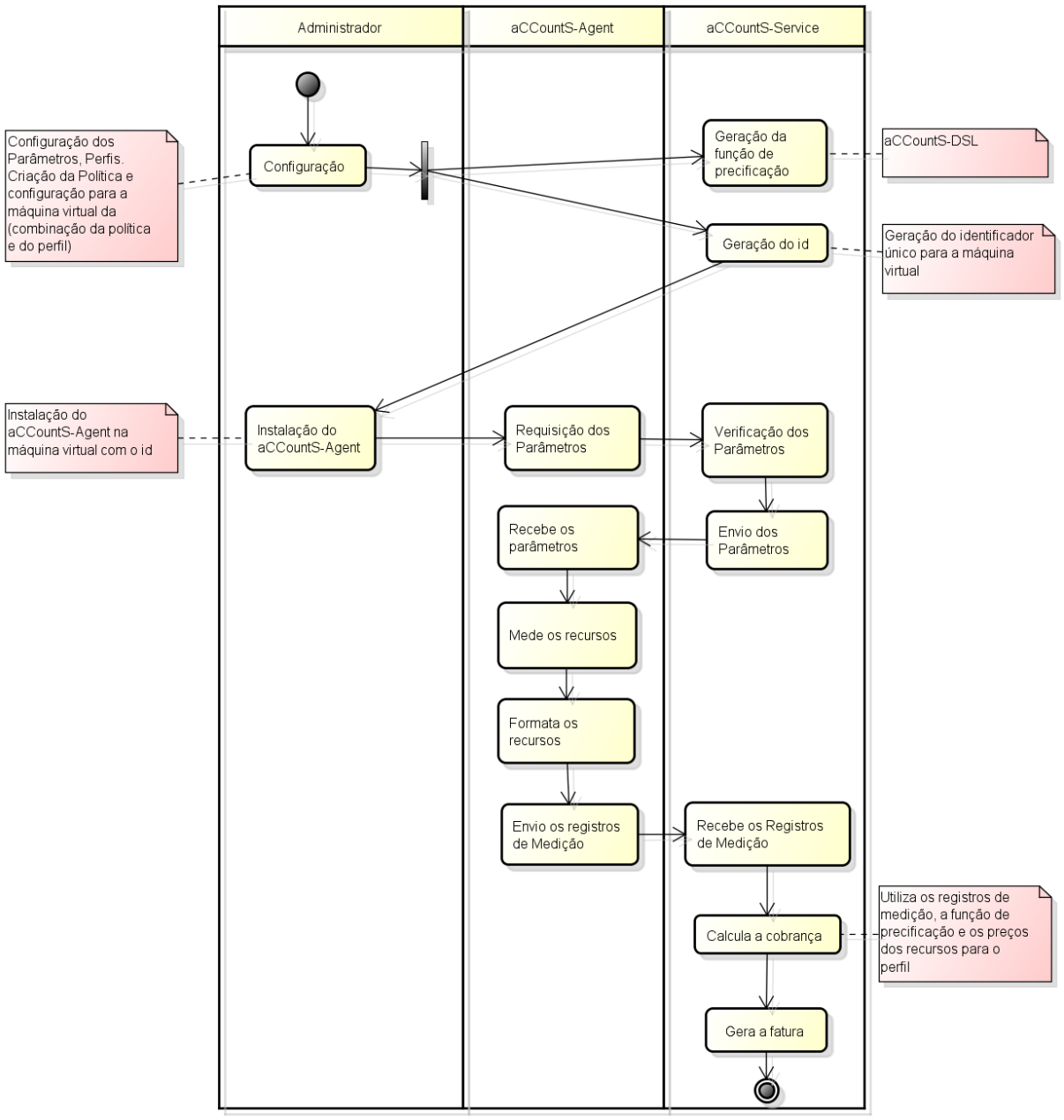

Figura 3.5: Diagrama de atividades do processo do *aCCountS*

Por meio desse diagrama é possível verificar que ao realizar as configurações no *aCCountS-Service*, definindo a política e a configuração da máquina virtual, com o perfil e a política específica, são gerados a função de precificação e o identificador da máquina virtual,

respectivamente. Ao gerar o identificador, o agente é instalado na máquina virtual configurado com ele. A partir disso, inicia-se o processo de busca pelos parâmetros a serem medidos, medição e a mediação desses recursos, que são enviados para o *aCCountS-Service*.

Em seguida serão mostrados com detalhes os processos que ocorrem nos macro-componentes *aCCountS-Agent* e *aCCountS-Service* e a comunicação entre eles.

3.1.1 APIs de comunicação entre os macro-componentes do *aCCounstS*

A comunicação entre o *aCCountS-Agent* e o *aCCountS-Service* é realizado por meio de troca de mensagens no formato JSON (JSON, 2013). Como ilustrado na Figura 3.4(b), o *aCCountS-Service* fornece uma *API* para comunicação com o agente. A interface *iParameters* serve para que o *aCCountS-Agent* obtenha do serviço, quais são os recursos (parâmetros) a serem monitorados e como estes devem ser obtidos da máquina virtual onde o agente está sendo executado. Já a interface *iResources* permite que o agente envie os registros de medição, a partir dos recursos monitorados. Dessa maneira, qualquer empresa de nuvem pode criar seu próprio agente para tarifar seus serviços, desde que mantendo a compatibilidade com a API fornecida.

Por exemplo, caso seja necessário monitorar o uso do processador ou consumo de memória, o serviço deve fornecer a forma como o agente pode obter os valores de tais recursos.

A *iResources* periodicamente envia ao *aCCountS-Service* as informações de uso da máquina virtual para o cálculo da fatura. Nas próximas duas sub-seções serão descritos em mais detalhes como os sub-componentes *aCCountS-Agent* e *aCCountS-Service* operam para o estabelecimento do fluxo de tarifação.

3.1.2 O Agente - *aCCountS-Agent*

O *aCCountS-Agent* é formado pelos sub-componentes *AgentReceptorManager*, o *MeteringVM*, *MediatorVM* e pelo *AgentSendManager*, como modelado pelo diagrama de componentes mostrado na Figura 3.4(a). Os componentes *MeteringVM* e *MediatorVM*, correspondentemente, implementam as funções do *Metering* e do *Mediator* do fluxo de contabilidade do *aCCountS*. Cada sub-componente do *aCCountS-Agent* realiza um passo para atender a funcionalidade de monitoramento de recursos de uma máquina virtual. Na Figura 3.6 é ilustrado o diagrama de classes do *Agent* com suas principais classes abaixo detalhadas.

- A classe *AgentReceptor* requisita do *aCCountS-Service* as regras (parâmetros) através da operação *getRules()*, a fim de permitir ao *Agent* realizar a medição. Ela possui os atributos: *(i) machine_id*, para identificar a máquina e assim, quais as variáveis (recursos) devem ser medidos na máquina virtual, *(ii) service_URL*, que armazena o caminho para o método no *aCCountS-Service*, que disponibilizará as regras para o monitoramento e *(iii)*, *rules*, que são as regras de medição formadas pelos recursos que serão medidos e os comandos para medi-los;
- A classe *Rule*, que herda da classe *Variable* e possui todas as suas características, possui

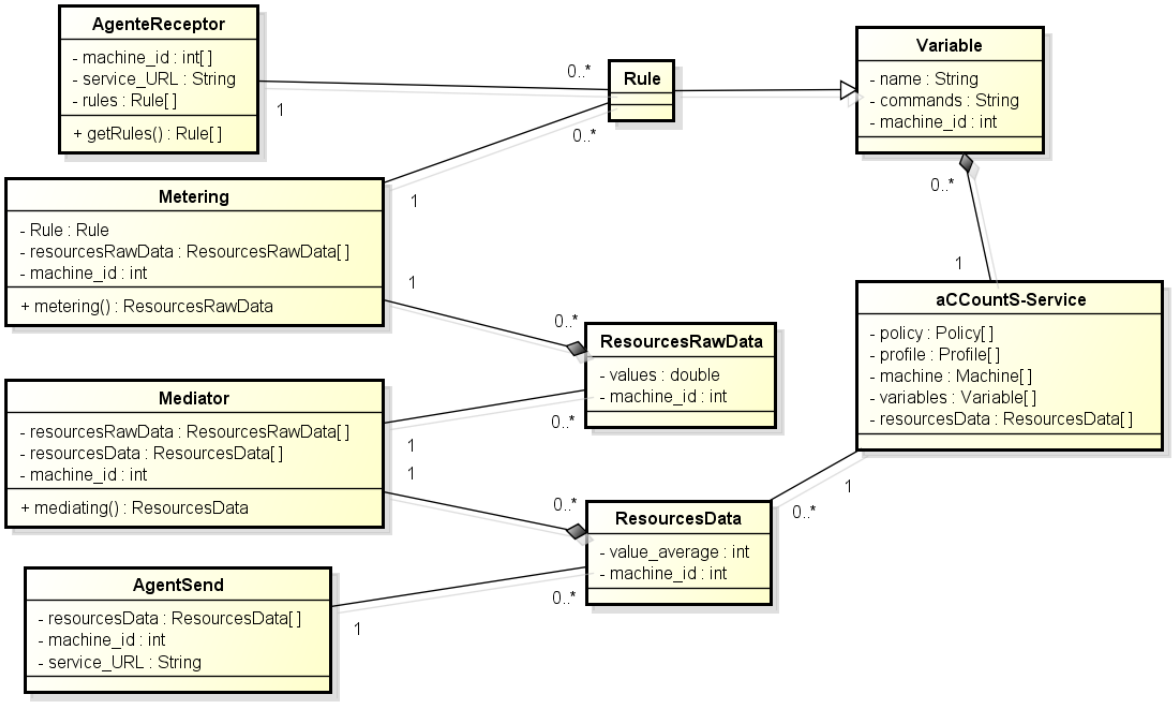

Figura 3.6: Diagrama de classes do *aCCountS-Agent*

como atributos: *(i) name*, o nome do recurso a ser medido, *(ii)*, *commands*, comando utilizado para medir o determinado recurso e *(iii) machine_id*, referência à máquina que contém as informações para tarifação;

- A classe *Metering* possui a função de medir o consumo dos recursos na máquina virtual. Ela realiza essa funcionalidade por meio da operação *metering* e precisa das regras para medir. Para isso, essa classe conta com os seguintes atributos: *(i) rule*, recurso e comando que usará para realizar a medição, *(ii) machine_id*, identificação da máquina, da qual os dados medidos estarão associados e *(iii) resourcesRawData*, recurso medido;

- A classe *ResourceRawData* possui os atributos *(i) values*, valor de consumo medido na máquina virtual e *(ii) machine_id*, para identificar de qual máquina pertence os recursos medidos;

- A classe *Mediator* possui a função de calcular a média dos recursos consumidos, a fim de tornar o processo de tarifação mais fácil. A classe possui a operação *mediating* para realizar a mediação dos recursos e os atributos: *(i) resourcesRawData*, dados medidos na máquina virtual que servirão de entrada para a operação de mediação, *(ii) resourcesData*, valor processado pela função de mediação a partir de vários *resourcesRawData* e *(iii) machine_id*, que vincula esses dados à maquina no serviço;

- A classe *ResourcesData* possui os atributos *(i) value_average*, média dos dados medidos na máquina virtual e *(ii) machine_id*, referência à máquina no serviço;

- A classe *aCCountS-Service* representa o componente *aCCountS-Service* com todas as suas características, seus atributos principais são: *(i) policy*, política de tarifação, *(ii)*

profile, perfil de hardware utilizado, *(iii) machine*, máquinas que combinam uma política e um perfil para tarifação de uma máquina virtual, *(iv) variable*, dados criados para serem monitorados no agente e *(v) resourcesData*, dados medidos e processados no agente que serão utilizados para o cálculo da fatura.

A partir desse diagrama de classes foram definidos os diagramas de sequência, que mostram o fluxo de chamadas para realizar o processamento dentro do agente. Na Figura 3.7, tem-se o diagrama de sequência da solicitação de parâmetros pelo agente para o serviço.

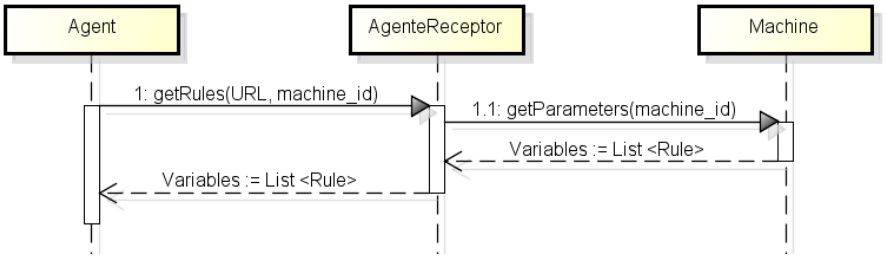

Figura 3.7: Diagrama de sequência de requisição das regras de medição pelo *aCCountS-Agent* para *aCCountS-Service*

Esse diagrama representa o processo de inicialização do serviço partindo do agente, no qual estão configurados o caminho para o serviço (URL) e a máquina associada à máquina virtual. Essa máquina é responsável por disponibilizar quais dados precisam ser medidos para o cálculo do faturamento. O *Agent*, portanto, ativa a classe *AgentReceptor* por meio da chamada *getRules(URL, machine_id)*, requisição das regras ao serviço, que por sua vez recebe as regras do serviço (máquina) através da requisição *getParameters(machine_id)*. Por fim, uma lista de regras é retornada ao *AgentReceptor* e ao *Agent*. O diagrama de sequência ilustrado na Figura 3.8 representa o fluxo de medição e mediação no *Agent*.

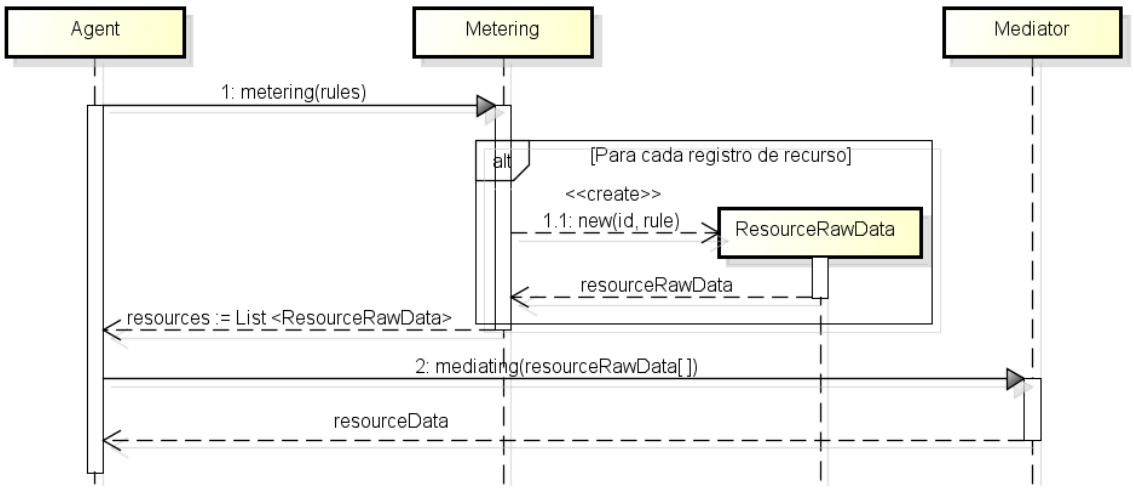

Figura 3.8: Diagrama de sequência do fluxo de medição e mediação no *aCCountS-Agent*

Nesse diagrama o *Agent* solicita à classe *Metering* a medição dos recursos por meio da operação *metering*, que utiliza as regras recebidas para realizar essa funcionalidade. Então

para cada regra definida para um recurso é realizada a medição e gerado um objeto *Resource-RawData*, que representa a agregação dos dados crus medidos na máquina virtual. A classe *Metering*, após todas as medições, retorna para o *Agent* uma lista de *ResourceRawData*.

Com os registros de medição, o *Agent* requisita à classe Mediator a mediação dos recursos por meio da operação *mediating*. Essa operação recebe como parâmetro uma lista de *resourceRawData*, calcula a média de utilização dos recursos e retorna para o *Agent* o registro de medição tratado, o *resourceData*. Para finalizar o processo dentro do agente, o diagrama de sequência (Figura 3.9) representa o processo de envio dos dados monitorados para o *aCCountS-Service*.

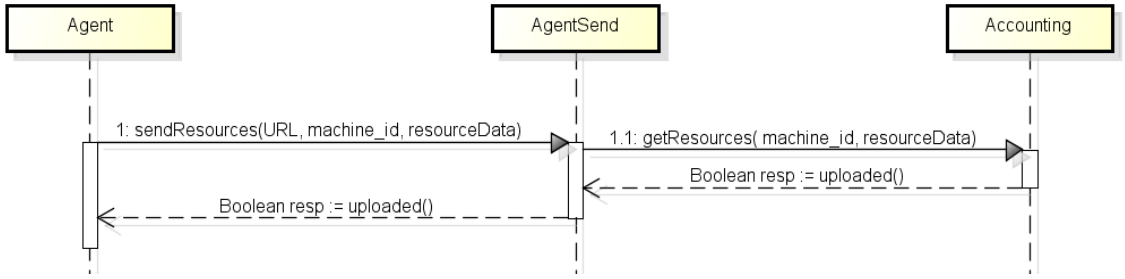

Figura 3.9: Diagrama de sequência de envio dos dados do *aCCountS-Agent* para o *aCCountS-Service*

No diagrama da Figura 3.9, a chamada ao método *sendResources(URL, machine_-id, resourceData)* da classe *AgentSend*, permite que o *Agent* envie os recursos medidos para o serviço. Estes dados são repassados pela classe *AgentSend* para a classe *Accounting*, por meio do método *setResources(machine_id, resourcesData[])*. Este método permite que a partir da identificação de uma máquina, seus dados sejam armazenados no *Accounting*. Ao receber os dados com sucesso, o método *uploaded(sucesso)* é ativado na classe *Accounting*, enviando uma resposta de sucesso ao agente. Em caso de falha, haverá necessidade de reenvio dos dados.

Por meio do diagrama de componentes 3.4(a), na arquitetura do *aCCountS-Agent*, o *AgentReceptorManager* é responsável por buscar no *aCCountS-Service* os parâmetros (interface *iParameters*), interpretá-los e enviá-los ao sub-componente *MeteringVM*. Um *AgentReceptor-Manager* de uma máquina virtual está associado a uma máquina no *aCCountS-Service* por meio de um identificador único (*id*). Na requisição dos parâmetros, o agente utiliza este identificador, permitindo ao *aCCountS-Service* reconhecer a política de tarifação da máquina virtual e quais recursos que devem ser monitorados, para então, fornecê-los de forma correta ao agente, de acordo com seu sistema operacional. Cada parâmetro recebido pelo *AgentReceptorManager* é um arquivo JSON com uma lista dos recursos e seus respectivos comandos para executar o monitoramento na máquina virtual. Dado que os *(i)* parâmetros e seus comandos de execução, *(ii)* a política definida para tarifação da máquina virtual e (*iii*) seu perfil de precificação foram definidos na primeira parte do processo, a etapa de configuração do *aCCountS* está completa.

Por meio dos comandos passados pelo *AgentReceptorManager*, o *MeteringVM* mede a utilização de cada recurso na máquina virtual em uma certa frequência predeterminada e envia esses registros para o componente *MediatorVM* que os intermedia. O *MediatorVM* recebe os dados medidos, calcula a média de utilização dos recursos e a cada hora, gera um registro de

medição. Esses registros são conduzidos para o *AgentSendManager*, que os expede, uma vez ao dia, para o *aCCountS-Service* (API *iResources*).

Os tempos de medição e de envio dos registros ao serviço podem ser configurados pela administrador da infraestrutura de nuvem, mas a configuração padrão no *MeteringVM* indica que os dados são monitorados a cada minuto, enquanto o *MediatorVM* realiza suas funções em intervalos de uma hora. Esta decisão pretende não sobrecarregar a rede com informações de medição a cada minuto no tráfego de dados entre agente e serviço, nem o próprio *aCCountS-Service*. O *aCCountS-Service* com posse das informações de consumo de uma máquina virtual inicia seu processo de tarifação por meio dos seus sub-componentes, como especificado na próxima sub-seção. Para melhor o entender o processo de tarifação dentro do agente, a Figura 3.10 representa o diagrama de atividades do *aCCountS-Agent*.

O agente é instalado na máquina virtual e executado em cima do sistema operacional, dessa forma os comandos de medição dos recursos enviados devem ser compatíveis com o sistema operacional hospedeiro. As atividades que ocorrem no *AgentReceptorManager* são de comunicação com o *aCCountS-Service* a fim de obter os parâmetros para o monitoramento dos recursos na máquina virtual. O *MeteringVM* recebe a informação de quais recursos deverão ser medidos e os comandos para medi-los e envia a cada segundo um registro de medição para o *MediatorVM* que formata esses dados recebidos. Por exemplo, o uso dos recursos (CPU, memória, armazenamento), no qual a medição é realizada em porcentagem é calculado a média de uso a cada minuto. As transações ocorridas, como *uploads* e *downloads*, são recebidas a cada segundo e somadas gerando um valor total, tornado os dados num formato mais fácil de ser contabilizado. Por fim, o *AgentSendManager* recebe os registros formatados e os armazena até enviá-los, uma vez ao dia.

3.1.3 O Serviço - *aCCountS-Service*

O serviço é o responsável por realizar a tarifação de forma independente de infraestrutura de nuvens, recebendo os dados de consumo das máquinas virtuais, ele os processa de forma a definir a fatura do cliente por meio do perfil e das políticas definidas. A Figura 3.11 representa o diagrama de classes desse componente, cujas classes serão detalhas em seguida.

- A classe *Agent* é responsável por receber os dados a serem medidos e enviar os dados consumidos na máquina virtual. Nesse diagrama apenas seus atributos foram representados para visualizar os dados que a classe comporta, são eles: *(i) operationSystem*, sistema operacional hospedeiro da máquina virtual, *(ii) rules*, regras utilizadas para medição do consumo na nuvem, *(iii) resourcesRawData*, dados de consumo medidos na máquina virtual, *(iv) resourcesData*, dados processados no agente, *(v) machine_id*, identificação da máquina;
- A classe *ResourcesRawData* possui os atributos *(i) values*, valor de consumo medido na máquina virtual e *(ii) machine_id*, para identificar de qual máquina pertence os dados medidos;

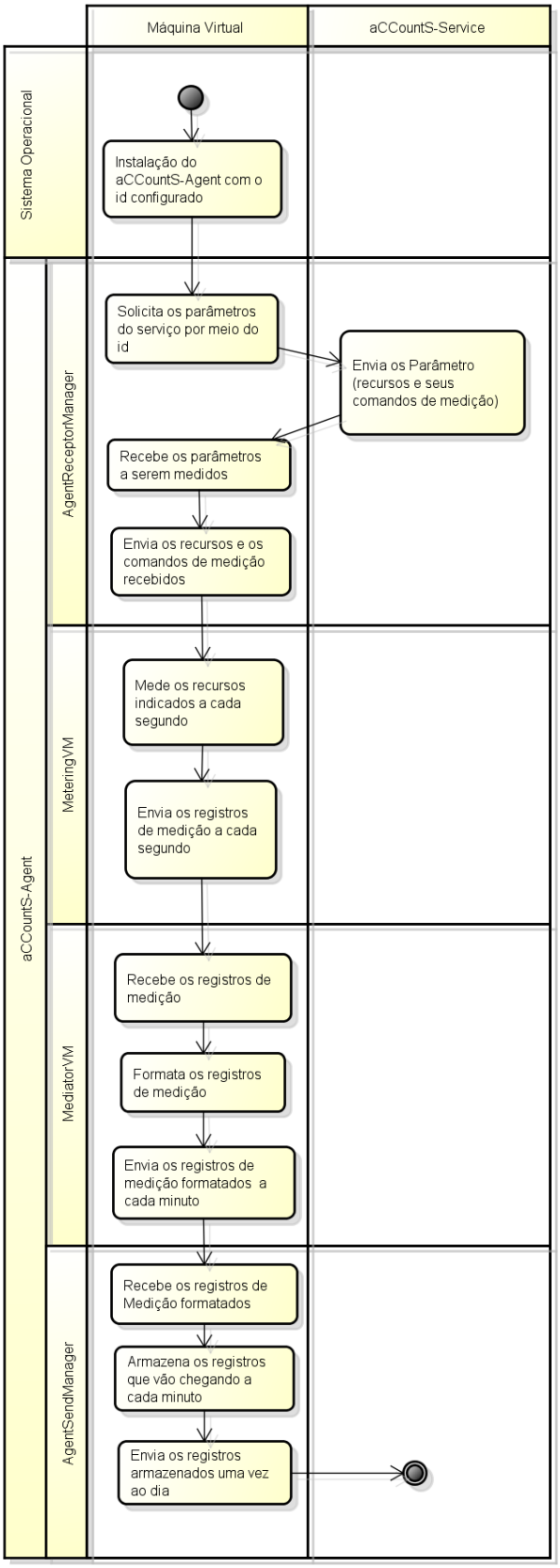

Figura 3.10: Diagrama de atividades do processo do *aCCountS*

- A classe *(ResourcesData)* possui os atributos *(i) value_average*, média dos dados medidos

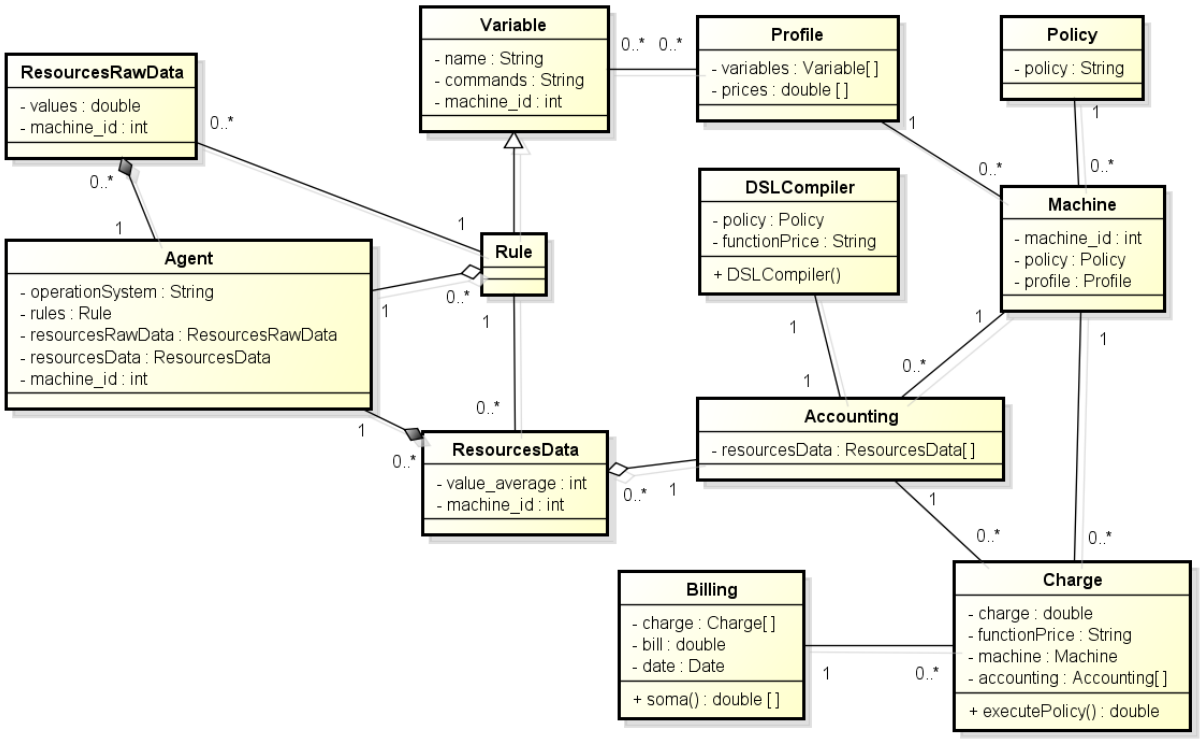

Figura 3.11: Diagrama de classes do *aCCountS-Service*

na máquina virtual e *(ii) machine_id*, referência à máquina no serviço;

- A classe *Variable*, possui como atributos: *(i) name*, o nome do recurso a ser medido, *(ii)*, *commands*, comando utilizado para medir o determinado recurso e *(iii) (machine_id)*, referência à máquina que está associada no serviço. A classe *Rule* herda dessa classe e possui todas as suas características;

- A classe *Accounting* possui a função de receber os dados do agente e ativar a função de precificação na classe *Charge*;

- A classe *Profile* tem a função de armazenar o perfil de hardware, variáveis que serão tarifadas e seus preços correspondentes. Para isso, possui os atributos: *(i) variables*, pois possui um subconjunto de variáveis definidas na classe *Variable* e *(ii) prices*, para cada variável é determinado um preço correspondente;

- A classe *Policy* tem a função de armazenar as políticas definidas pelo administrador e possui o atributo *policy*, política de tarifação apresentada por meio de um código definido na linguagem *aCCountS-DSL*;

- A classe *Machine* responsável por associar uma política e um perfil para a tarifação de uma máquina virtual. Ela é utilizada para associar as regras, variáveis e dados de consumo a uma configuração de política/perfil. Para isso, essa classe possui como atributos: *(i) machine_id*, identificador da máquina, *(ii)* policy, política de tarifação e *(profile)*, perfil de hardware e preços dos recursos;

- A classe *DSLCompiler* é responsável por compilar a política por meio do método *DSLCompiler*, que possui o compilador da linguagem. Os atributos dessa classe são: *(i) policy*, a política que será compilada e *(ii) functionPrice*, a função de precificação, resultado da compilação da política;

- A classe *Charge* utiliza *(i)* a função de precificação, *(ii)* os dados consumidos e *(iii)* os preços dos recursos para realizar o cálculo da cobrança por meio da operação *executePolicy()*. Essa classe possui como atributos: *(i) charge*, variável que guarda o valor do processamento do *executePolicy()*, *(ii) functionPrice*, função de precificação, *(iii) machine*, máquina, que por meio dela obtém-se qual a política que deve ser compilada e qual o perfil utilizado para definição dos preços dos recursos e *(iv) accounting*, dados que serão utilizados na função de precificação para calcular a cobrança;

- A classe *Billing* utiliza os dados de cobrança e somá-os por meio do método *soma()* ao fim de um período, calculando o custo da fatura. Para isso, essa classe possui os atributos: *(i) charges*, que armazena as cobranças calculadas na classe *Charge* para realizar o faturamento, *(ii) bill*, que guarda o valor da fatura final calculada em uma certa data e *(iii) date*, data do cálculo da fatura.

Para melhor entender o processo do *aCCountS-Service*, a Figura 3.12 ilustra seu diagrama de sequência.

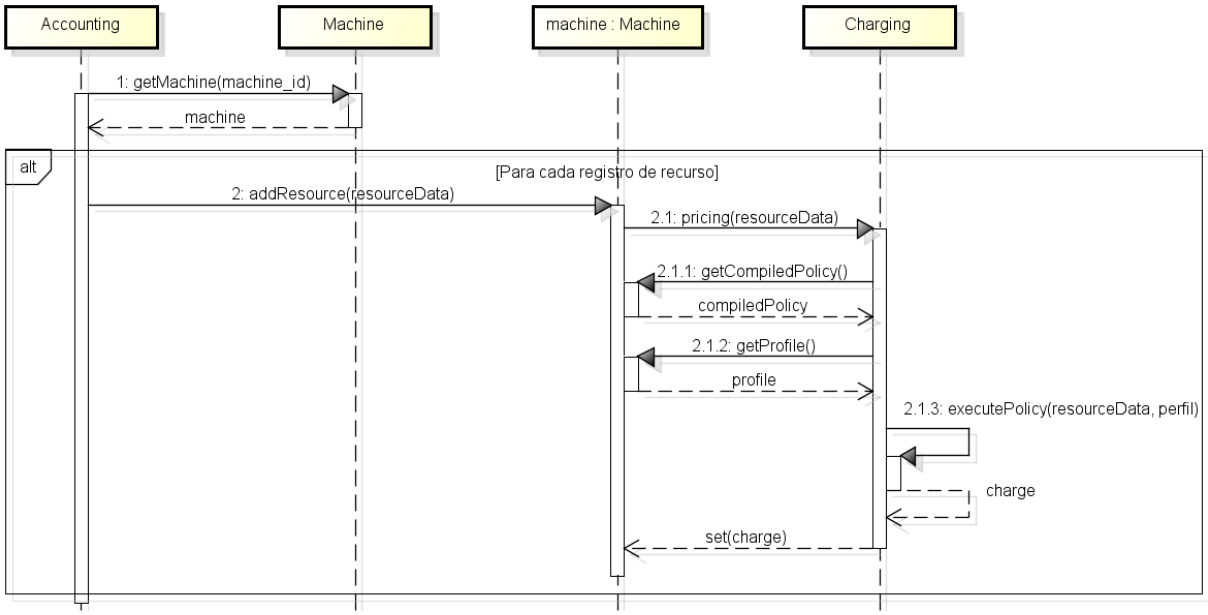

Figura 3.12: Diagrama de sequência do *aCCountS-Service*

Nesse diagrama, a classe *Accounting*, com uma lista de *resourceData* recebida do *Agent*, solicita à classe *Machine*, por meio da operação *getMachine*, a máquina, cujo identificador é passado como parâmetro. Essa classe, então, retorna ao *Accounting*, o objeto *machine*, ou seja a máquina correspondente. O *Accounting*, portanto, para cada *resourceData* da lista recebida do *Agent*: *(i)* adiciona o *resourceData* no objeto *machine* através da operação *addResource*; *(ii)* o objeto *machine* solicita à classe *Charging* o cálculo da cobrança, por meio

da operação *pricing* que passa como parâmetro o *resourceData*; *(iii)* a classe *Charging* chama o método *getCompiledPolicy* da classe *machine*, que retorna a política compilada, ou seja, a função de precificação. *(iv)* A classe *Charging* também solicita ao objeto *machine*, o perfil da máquina por meio da operação *getProfile*, e *(v)* a classe *Charging*, então, por meio da função de precificação e de posse dos recursos medidos pelo agente e do preço dos recursos, obtidos do perfil da máquina, realiza a operação *executePolicy*, gerando o valor da cobrança, *charge*. Por fim, *(vi)* a classe *Charging* define o *charge* no objeto *machine* que armazena esse valor e relaciona ao *resourceData* correspondente.

Como mostra o diagrama de componentes 3.4(c), a tarifação de cada máquina virtual é processada por meio dos sub-componentes do *aCCountS-Service*, que recebem os registros de medição e geram uma fatura mensal para o cliente. Esses sub-componentes são o *ResourcesManager*, o *AccountingManager*, o *PriceManager*, o *ProfileManager*, o *Machine-Manager*, o *BillingManager* e o *ClientManager*. O *PriceManager* é ainda subdivido em outros três componentes, a saber: *PolicyManager*, *DSLCompiler* e *ChargeManager*. A arquitetura do *aCCountS-Service* é ilustrada na Figura 3.4(c).

No *aCCountS-Service*, o *ResourcesManager* tem a função de enviar os parâmetros para o agente, quando solicitado. Duas entidades são manipuladas por este componente: o *Resources* e o *CommandsResources*. A primeira representa os recursos a serem monitorados, enquanto a segunda, os comandos que os agentes devem executar para medir tais recursos. Esta configuração deve ser feita pelo administrador da provedora da nuvem, que cadastra todos os recursos que poderão ser monitorados por ela e os seus respectivos comandos de execução. No protótipo desenvolvido, esta configuração é feita por meio de uma interface Web. O *Resources-Manager* ao ser consultado pelo agente, recebe o identificador da máquina virtual. Por meio deste identificador, sua política de tarifação é recuperada. Com isto, o *ResourcesManager* verifica na política quais recursos precisam ser medidos e os envia por meio de parâmetros (recursos e comandos) ao agente.

Já o *AccountingManager* tem a função de receber os registros de medição do agente e ativar o componente *PriceManager* para contabilizar os registros recebidos. Este componente é responsável por gerar a função de precificação para contabilizar os registros de medição, além de calcular o custo do serviço e gerar as cobranças. Estas tarefas são distribuídas entre seus sub-componentes: *PolicyManager*, *DSLCompiler* e *ChargeManager*.

O *PolicyManager* gerencia a criação das políticas de tarifação. Essas políticas são definidas através da DSL proposta nesse trabalho que será descrita com mais detalhes na subseção 3.1.4. O *DSLCompiler* compila a política de tarifação definida e gera uma função de precificação, enquanto o *ChargeManager* utiliza a função definida para calcular os registros de cobrança por meio dos recursos monitorados no agente e dos preços dos recursos (requisitados do componente *ProfileManager*).

O componente *ProfileManager* é responsável por gerenciar o perfil das máquinas virtuais do agente. Ele manipula duas entidades: o *Resources* que, como já detalhado anteriormente, representa cada recurso, e o *Price* que representa os preços desses recursos. Estes componentes são cadastrados pelo administrador da provedora de nuvem durante o processo de

configuração do serviço. No *ProfileManager*, além dos recursos e seus preços, deve-se cadastrar os perfis disponíveis na infraestrutura da nuvem, definindo a quantidade de memória, armazenamento em disco e tipo de processador das máquinas virtuais que podem ser instanciadas. De forma comparativa, a ideia de perfil na proposta segue a mesma proposição da *Amazon*.

O *MachineManager* tem a função de gerenciar a configuração das máquinas virtuais, definindo seu perfil e a política que a mesma será tarifada. Essa configuração gera um identificador, que associa a máquina virtual à sua política de cobrança e ao seu perfil.

Os registros de cobrança calculados pelo *ChargeManager* são enviados para o componente *BillingManager*, cuja principal função é somá-los para geração da a fatura que será enviada para o *ClientManager* ao final de cada mês, caso o modelo de tarifação seja pós-pago. Caso o modelo da política de tarifação seja pré-pago, uma mensagem é enviada para a provedora da nuvem no momento em que o total de créditos do cliente for esgotado. O valor do crédito é medido pelo agente e enviado para o *AccountingManager* via *iResources*. Com essa informação a provedora da infraestrutura de nuvem pode tomar providências pela continuidade ou não da oferta de serviços ao cliente cujo crédito se esgotou.

Para melhor visualizar o processo de tarifação no *aCCountS-Service*, a Figura 3.13 ilustra seu diagrama de atividades, mostrando o fluxo das atividades.

O primeiro passo no processo do *aCCountS-Service* é a criação dos perfis de máquinas e das políticas. Em seguida o administrador cria combinações de perfis e políticas, as quais servirão para tarifar as máquinas virtuais. Cada combinação é reconhecida por um identificador único , o id, e representa a política e o perfil de como uma máquina virtual será tarifada. Com o id, o agente pode ser instalado na máquina virtual e realizar o fluxo de contabilização do *aCCountS*.

Então, o administrador inicia definindo os parâmetros que podem ser monitorados na nuvem, por meio do componente *ResourcesManager*, cria os diferentes perfis de máquina e define os preços dos seus recursos, por meio do componente *ProfileManager* e implementa as políticas de tarifação suportadas pela provedora, por meio do componente *PolicyManager*. Com isso, o administrador cria as possíveis formas de tarifar as máquinas virtuais, por meio de combinações de perfis e políticas, no componente *Machine*. Essa combinação gera um identificador utilizado pelo administrador para configurar o agente. O agente, então, solicita ao serviço os recursos que serão monitorados identificando o id associado a ele. Esse id permite o *ResourcesManager* detectar quais os recursos precisarão ser monitorados, baseado naqueles declarados na política de tarifação associada ao id. Os parâmetros são então enviados e medidos pelo *aCCountS-Agent*, que retorna ao *AccountingManager* os registros de medição. Esse último ativa o componente *ChargeManager*, enviando esse dados recebidos, que solicita à *DSL-Compiler* a função de precificação associada ao id da máquina virtual do processo e os preços dos recursos do perfil associado ao id, por meio do *ProfileManager*. Em posse desses dados, a função de precificação calcula a cobrança de cada registro recebido diariamente e envia para o componente *BillingManager* que armazena os registros de cobrança por um tempo determinado, normalmente um mês, ou quando solicitado o calculo parcial da conta. Ao final do período os registros de cobrança são medidos e se o modelo de cobrança do cliente for pós-pago, sua fa-

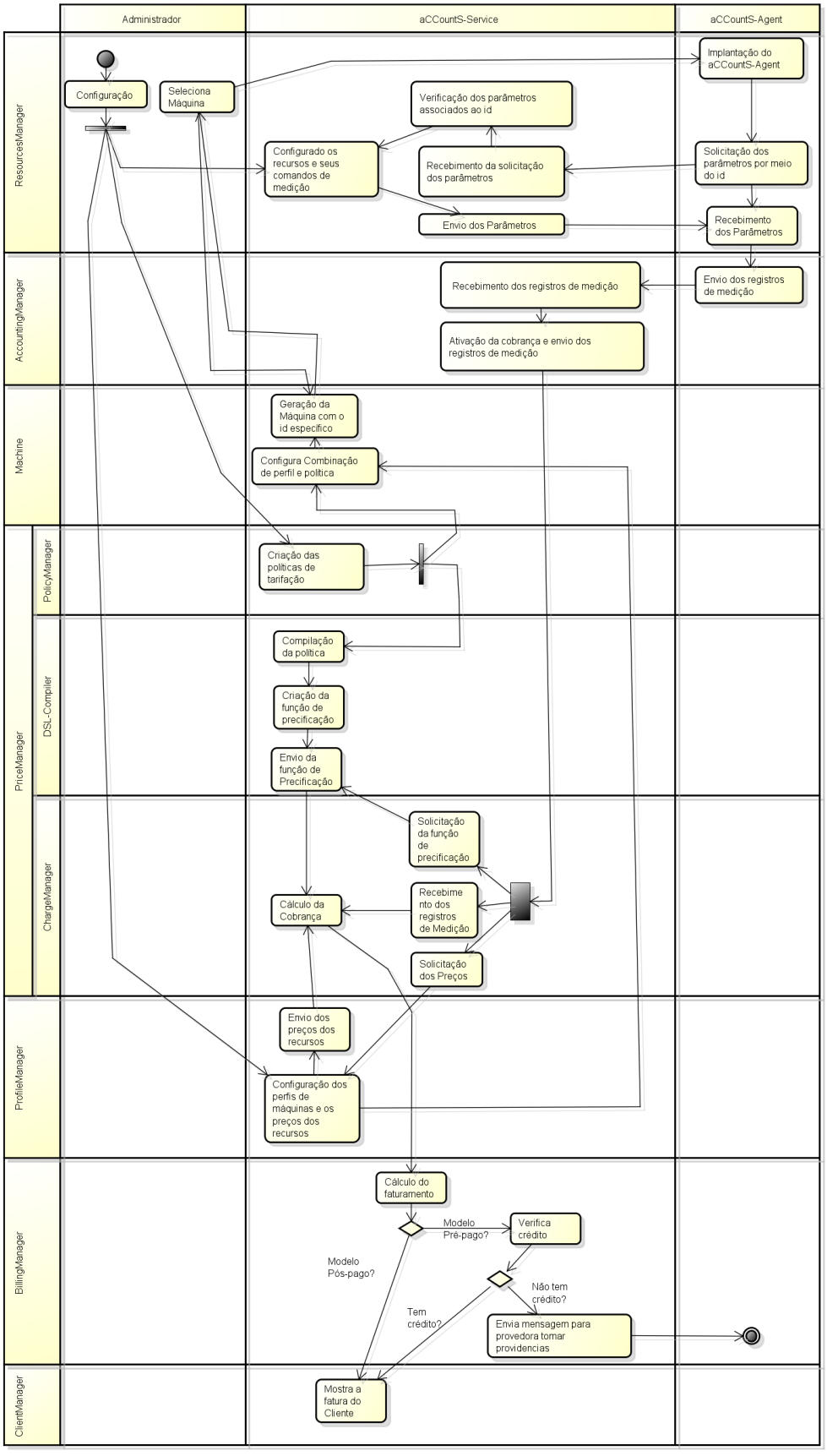

Figura 3.13: Diagrama de atividades do processo do *aCCountS*

tura é exibida, se não é verificado o crédito e debitado, se o crédito esgotou com transação ou se o valor é inferior a uma porcentagem do valor mínimo por exemplo, 10% de R$ 50,00) uma mensagem é enviada para a provedora de nuvem; se não, a fatura do cliente é gerada.

No processo de contabilização do *aCCountS-Service*, o administrador cria as políticas de tarifação para definir a função de precificação utilizada para gerar os registros de cobrança diários. Essas políticas são definidas em uma DSL proposta nesse trabalho e explicada com mais detalhes na subseção a seguir.

3.1.4 A DSL de tarifação - *aCCountS-DSL*

No contexto deste trabalho, uma política de tarifação é definida como um conjunto de regras que estabelece a forma como recursos das máquinas virtuais de um cliente são tarifados. A *aCCountS-DSL* é uma linguagem de domínio específico textual para criação de políticas de tarifação em nuvens de *IaaS* dentro da proposta *aCCountS*.

O projeto da *aCCountS-DSL* foi elaborado a partir de requisitos obtidos dos trabalhos relacionados previamente descritos no capítulo 2, bem como a partir de exemplos da indústria, com destaque para a *Amazon EC2*, uma vez que esta se trata da principal fornecedora de serviços de infraestrutura em nuvem do mundo. Para ilustrar esta influência, na etapa de configuração (primeira parte do processo de tarifação), o perfil da máquina é definido no *aCCountS-Service* pelo administrador, que precifica os recursos que serão contabilizados. Um conjunto de requisitos de tarifação determina o valor de um perfil, muito semelhante ao que ocorre na *Amazon EC2*, em que os diferentes requisitos definem as diferentes instâncias e seus preços. A *aCCountS-DSL* busca permitir que qualquer nuvem possa definir suas regras e criar sua política de tarifação através de uma linguagem que atenda aos requisitos utilizados pela *Amazon* e aos requisitos apontados nos trabalhos relacionados apresentados no Capítulo 2. Os requisitos utilizados pelo *aCCountS* são apresentados a seguir.

- *Garantia de alocação*: Algumas políticas de tarifação promovem maior garantia à nuvem do que outras, impactando nos custos dos seus recursos. A *Amazon* utiliza três modelos e os mesmos são suportados pela linguagem e serviço propostos. São eles: *(i) sob-demanda*, que permite ao cliente pagar pela capacidade computacional utilizada por hora, sem nenhum compromisso de longo prazo (AMAZON, 2013), *(ii) Reservado*, que oferece a opção de um pagamento único para cada instância que o cliente deseja reservar e, em troca, um desconto significativo sobre a custo de utilização dos serviços para essa instância (AMAZON, 2013), e *(iii) Lance*, que é concedido por meio de leilão dos recursos subutilizados, sem uma garantia sobre o tempo de disponibilidade do serviço.

  Este requisito não é definido na *aCCountS-DSL*. Para que o serviço o suporte é necessário defini-lo na etapa de configuração. Ao criar os perfis das máquinas, o administrador define perfis para políticas sob-demanda, reservada (1 ano, 2 anos etc.) e lance, ou seja, os preços dos recursos para políticas sob-demanda são maiores, que para políticas reservadas, que por sua vez, são maiores que para políticas lance. Por outro lado, os cliente contarão com as características que cada uma provê por meio da infraestrutura de nuvem.

- *Modelo de pagamento*: Os modelos propostos pela *Amazon* atualmente são pós-pagos. Entretanto, outros serviços de tecnologia aumentaram sua utilização através de planos pré-pagos, como a telefonia móvel e a Internet móvel (DANTAS, 2002). Dessa maneira, a linguagem proposta reconhece dois modelos possíveis de pagamento. O primeiro é o modelo *pré-pago*, no qual um cliente paga antes de utilizar os recursos, podendo limitar seu uso pelos créditos comprados. Já o modelo *pós-pago* reflete o modo tradicional em que um cliente contrata um plano de utilização e paga após o uso, em geral numa frequência mensal. Estes modelos foram influenciados pela proposta de Elmroth et al. (ELMROTH et al., 2009) e pelos serviços de tecnologia citados anteriormente.

  O modelo de pagamento é suportado pela arquitetura proposta, entretanto ainda não está completamente funcional no protótipo implementado. A aplicação desse requisito acontece em quatro etapas, sendo que as três primeiras ocorrem no momento de configuração do serviço, enquanto a quarta ocorre durante a execução do mesmo. A primeira etapa acontece na configuração dos parâmetros, em que o administrador define o recurso crédito, e o associa a um determinado cliente. A segunda etapa ocorre quando da criação dos perfis das máquinas, e leva em consideração o modelo da política. No caso, definem-se preços maiores para perfis com modelo pré-pago. A terceira etapa permite ao administrador configurar os tempos de medição e de envio de dados para o serviço. Já na última etapa, o agente faz a medição do recurso de crédito. Assim como no requisito *Garantia de Serviço*, esse requisito não é definido por meio da linguagem *aCCountS-DSL*.

- *Forma de contabilização*: Segundo alguns autores (I.R.; Y.K.; P.G., 2006), (CARACAS; ALTMANN, 2007), (ELMROTH et al., 2009), (NARAYAN et al., 2012), um serviço em nuvem consome diferentes recursos e em quantidades diferentes, sendo portanto justo pagar apenas pelos recursos consumidos, evitando que aqueles que utilizem pouco paguem o mesmo que outros que sobrecarregam o sistema. Desta forma, este requisito estabelece que um cliente possa ser tarifado por uso dos recursos (ex.: uso de 10% de CPU de uma máquina virtual em uma hora custa $ 0.003) ou por tempo de uso (1h de uso de uma máquina custa $ 0.3) ou ainda por uma opção híbrida definida pela provedora de nuvem, podendo então o administrador definir as regras da política da forma em que melhor lhe convier. Esse requisito utiliza a *aCCountS-DSL* para ser definido, através da criação de regras de cobrança, calculando os custos dos recursos por meio de operações aritméticas, como será detalhado posteriormente.

- *Perfil de hardware*: máquinas com diferentes capacidades de processamento e disponibilidade de armazenamento e memória apresentam preços diferentes. Este requisito é suportado pela arquitetura por meio da configuração dos perfis das máquinas. O administrador deve levar em consideração o tipo de hardware para precificar os recursos do perfil.

  Assim como os requisitos *Garantia de alocação* e *Modelo*, esse requisito não é definido na *aCCountS-DSL*. Ele é suportado pela arquitetura e sistema propostos, ao configurar o serviço de tarifação, definindo preços para os recursos de acordo com seu perfil de hardware. Na criação dos perfis das máquinas, o administrador leva em consideração o hardware disponível na nuvem, definindo diferentes perfis para atender várias demandas

dos clientes. Pode-se exemplificar perfis no *aCCountS* como um perfil Pequeno, Médio ou Grande. Dependendo da capacidade de hardware fornecido por esse perfil seus preços são maiores ou menores.

- *Utilitários*: Na configuração de uma máquina virtual para um cliente, recursos adicionais podem ser associados à tarifação. Por exemplo, clientes podem ser tarifados pelo uso de software proprietário (como o sistema operacional Microsoft Windows), por serviços disponibilizados pela provedora de nuvem (como elasticidade automática) ou por uso de centro de dados localizados em regiões estratégicas com maior disponibilidade ou recursos adicionais. Por exemplo, no centro de dados da *Amazon EC2* localizado na Virgínia (EUA), uma instância *small* utilizando o Linux custa $ 0.06 por hora, enquanto no centro de dados da Amazon situado em São Paulo, a mesma instância custa $ 0.08 por hora. Em geral, software, serviços e *datacenters* requerem custos adicionais à provedora de nuvem, sendo portanto, requisitos a serem considerados na definição da política de tarifação. Esse requisito é definido através da *aCCountS-DSL* ao definir as regras da política por meio de operações aritméticas que implementam instruções de cobrança desse utilitários consumidos pelo cliente.

- *Bem Estar social*: Este conceito relaciona-se com a ideia de se promover descontos na fatura do cliente em determinadas situações, como o uso de máquinas virtuais configuradas visando a economia de energia (YU; BHATTI, 2010), grande quantidade de recursos utilizados por um cliente específico (I.R.; Y.K.; P.G., 2006), recursos da nuvem sub-utilizados (CARACAS; ALTMANN, 2007) e ainda questões que atendam aos SLAs (multa às provedoras, quando houver alguma violação nesses acordos) (I.R.; Y.K.; P.G., 2006). Esses requisitos são definidos nas regras de cobrança.

  Esse requisito visa incentivar os clientes e provedoras de nuvens a atitudes que respeitam o meio ambiente e as relações sociais. O mesmo é definido através da linguagem *aCCountS-DSL* por meio das regras definidas na política utilizando operações aritméticas e comandos relacionais.

A partir da definição dos requisitos e de como é fornecido o suporte aos mesmos, a flexibilidade da tarifação é alcançada pela combinação das várias possibilidades de tarifação em políticas que possam atender diferentes tipos de clientes (com necessidades, recursos e perspectivas diferentes) e atender a diferentes objetivos de cobrança por parte das provedoras de nuvem (lucro, bem estar social e preços justos).

O conjunto de requisitos apresentado serviu de base para a definição da linguagem *aCCountS-DSL*. Através dela pode-se definir diferentes conjuntos de regras para atender aos diferentes requisitos propostos. Eles podem ser combinados entre si, criando uma sequência de instruções (regras) para definir a política de tarifação. De modo a ilustrar seus principais elementos, o Código 3.1 apresenta a estrutura geral de uma política descrita usando a linguagem. Ressalta-se que seu objetivo maior é a flexibilidade na definição das políticas de tarifação, de modo a utilizar diferentes recursos das máquinas virtuais de uma nuvem de infraestrutura, que possam ser medidos e precificados.

Código 3.1: Definição de uma política de tarifação com a *aCCountS-DSL*

```
1  Policy nome_da_politica [extends outra_politica] {
2    var {
3      definicao das variaveis auxiliares para
4      definir regras de negocio;
5    }
6    rules{
7      definicao de regras de negocio atraves
8      de operacoes aritmeticas e logicas;
9    }
10   return valor_da_cobranca;
11 }
```

Os elementos que estruturam a política são definidos da seguinte forma: O elemento ***policy*** representa o nome de uma política específica. Este elemento divide-se em duas seções. Na primeira seção, especificada pela palavra-reservada ***var***, são definidas variáveis auxiliares que servirão para compor as regras de tarifação. Como restrição da linguagem, as variáveis utilizadas só podem guardar valores numéricos de ponto flutuante, uma vez que as políticas tipicamente trabalham com manipulação de valores deste tipo. A segunda seção, chamada ***rules***, define as regras de composição da política de tarifação por meio de atribuições, operações aritméticas (soma, subtração, divisão e multiplicação) sobre variáveis e valores reservados na linguagem e pelo comando de seleção if, que executa determinadas regras, caso a condição definida para esse comando for verdadeira. As regras definidas na linguagem representam as regras de negocio da política, e são definidas em função dos recursos medidos em cada máquina virtual. Ao final da definição de uma política, utiliza-se a palavra-reservada ***return***, seguida da variável que representa o cálculo final do custo para uma política de tarifação, em função dos valores calculados nas regras.

De modo a facilitar a definição de novas políticas, pode-se reutilizar uma política pré-definida, de modo semelhante a ideia de herança (especialização), bastante conhecida em linguagens de programação. De modo semelhante, variáveis e regras da política pré-definida são reaproveitadas, além de se permitir que novas regras e variáveis sejam introduzidas ou ainda sobreposição dos mesmos conceitos. Para isso, usa-se, após o nome da política, a palavra reservada ***extends*** e em seguida, a política que se quer estender.

Para melhor ilustrar o uso da *aCCountS-DSL*, o Código 3.2 apresenta a definição de uma política, intitulada *SobPeqUsoPosGreat*. Seu nome referencia alguns dos requisitos que ela atende, a saber: *Sob*-demanda (garantia de alocação), *Peq*uena (perfil de máquina), *Uso* (forma de contabilização), *Pos*-pago (modelo), *Great*, por contabilizar questões a mais (requisitos Utilitários e de Bem estar social). Nela são definidas onze variáveis para auxiliar a definição das regras: *taxaCentralDados*, *memoria*, *cpu*, *armazenamento*, *transacaoBD*, *upload*, *download*, *descontoEnergia*, *descontoUsuario*, *desconto* e *custo*.

Código 3.2: Política de tarifação criada por meio da *aCCountS-DSL*

```
1  Policy SobPeqUsoPosGreat {
2    var {
3      taxaCentralDados; memoria; cpu; custo; armazenamento;
```

```
4       transacaoBD; upload; download; descontoEnergia;
5       descontoUsuario; desconto;
6     }
7     rules{
8       descontoEnergia = 0;
9       descontoUsuario = 0;
10      taxaCentralDados = 0.14;
11
12      memoria = instance.memoria * $memoria;
13      cpu = instance.cpu * $cpu;
14      armazenamento = instance.armazenamento * $armazenamento;
15      transacaoBD = instance.transacaoBD * $transacaoBD;
16      download = instance.download * $download;
17      upload = instance.upload * $upload;
18
19      if (instance.economiaEnergia >= 0.5 ){
20        descontoEnergia = 0.04;
21      }
22
23      if (instance.economiaEnergia >= 0.8 ){
24        descontoEnergia = 0.08;
25      }
26
27      if (instance.memoria >= 0.8 ){
28        descontoUsuario = 0.05;
29      }
30
31      if (instance.armazenamento >= 0.8 ){
32        descontoUsuario = descontoUsuario + 0.05;
33      }
34
35      custo = memoria + cpu + armazenamento + transacaoBD + upload +
36            instance.software + instance.servico + taxaCentralDados;
37      desconto = instance.sla + descontoEnergia + descontoUsuario;
38      custo = custo - custo * desconto;
39    }
40    return custo;
41  }
```

Um aspecto fundamental para a política de tarifação é que a mesma precisa ter acesso aos valores definidos durante a configuração do serviço em relação aos recursos monitorados e seus respectivos preços. Para este propósito, a *aCCountS-DSL* utiliza dois elementos. O primeiro é representado pela expressão *instance.recurso*, que representa o valor de um recurso monitorado (por exemplo, percentual médio de uso memória) em uma instância de uma máquina virtual. O segundo é representado pelos valores definidos na precificação dos recursos, e que são acessados utilizando como prefixo o símbolo $. Esses valores são recuperados a partir dos componentes do *aCCountS-Service*.

O suporte à execução das políticas definidas na linguagem *aCCountS-DSL* é feita pelo *DSLCompiler*. Este componente atua como um compilador, verificando sintaticamente as

políticas e gerando versões das mesmas para outras linguagens, que então podem ser executadas.

No processo de geração de código para uma política de tarifação, o *DSLCompiler* entende que as ocorrências *instance* são recursos medidos na máquina virtual. Estes valores devem ser recuperados no *AccountingManager*, que através do nome da variável, retorna os dados dos recursos medidos em uma máquina virtual em um determinado período. Já as variáveis iniciadas com $ equivalem aos preços dos recursos e, por sua vez, devem ser obtidos no componente *ProfileManager*, através da entidade *Price*, que retorna o preço correspondente ao recurso identificado. Como detalhado na explanação sobre os componentes do *aCCountS*, os nomes definidos para os recursos são previamente configurados no componente *ResourcesManager*, e para simplificar os processos, sugere-se que os nomes utilizados para os recursos e seus respectivos preços sejam os mesmos.

Na política *SobPeqUsoPosGreat*, os valores de *instance.economiaEnergia* e *instance.sla* são medidos na máquina virtual. Eles representam porcentagens de economia de energia realizada por um usuário e de violação no acordo de serviço (tempo que a máquina ficou disponível), evidenciados na máquina virtual respectivamente. Esses valores são calculados com a finalidade de verificar as configurações das máquinas virtuais do usuário e o quanto elas promovem a economia de energia, baseado em análises de desempenho (IMADA; SATO; KIMURA, 2009) e, no caso do SLA, o quanto os acordos de qualidade de serviços são atendidos ou violados.

As variáveis de instância *instance.memoria*, *instance.cpu*, *instance.armazenamento*, *instance.transacaoBD*, *instance.upload* são medidas de processamento na máquina virtual. As variáveis *descontoEnergia* e *descontoUsuario* são definidas a partir de comandos de seleção declaradas na política, atendendo a requisitos relacionados ao Bem Estar Social e Utilitários. A *taxaCentralDados*, definida na política com o valor de $ 0.14 mensais, é um valor pago pelo uso das máquinas do centro de dados da provedora de nuvem e assim como outras variáveis podem ser definidas pelo administrador com valores definidos pelos mesmos para atender suas políticas de negócio.

O Código 3.3 exemplifica o caso da extensão de uma política pré-existente, em que é utilizado a palavra reservada *extends*.

Código 3.3: Política criada por meio da *aCCountS-DSL* usando *extends*

```
1  Policy Res2GranUsoPosGreat extends SobPeqUsoPosGreat{
2     var {
3      custo; taxaFixa;
4           }
5     rules{
6        taxaFixa = 0.001;
7        custo = custo + taxaFixa;
8     }
9     return custo;
10 }
```

Na política *Res2GranUsoPosGreat*, seu nome evidencia alguns dos requisitos que ela atende: *Res*ervada *2*ano (garantia de alocação), *Gran*de (perfil de máquina), *Uso* (forma

de contabilização), *Pos*-pago (modelo), *Great* (requisitos Utilitários e Bem estar social). Ela estende a política *SobPeqUsoPosGreat*, representada na Figura 3.2 e dessa forma reutiliza todas suas variáveis e regras. Com isso a política *Res2GranUsoPosGreat* contém doze variáveis, sendo onze da política estendida e a décima segunda, a variável local *taxaFixa*. Esta nova variável representa taxa paga pelo firmamento do contrato de 2 anos, que no contexto da política vale $ 0.001 mensais. A regra atribuída à variável custo é redefinida, e uma nova política é definida com poucas linhas de código.

Por meio da linguagem *aCCountS-DSL* uma empresa de nuvem pode criar várias políticas diferentes, combinando os distintos requisitos de tarifação. Essas empresas podem levar em consideração diferentes perfis de máquina (hardware), os serviços e sistemas disponíveis (sejam eles gratuitos ou comerciais) e outras questões, como já foi discutido nesse trabalho. Com várias políticas de cobrança diferentes disponíveis, é possível que uma ou mais atendam aos perfis dos clientes de nuvem. Dessa forma, pode-se definir políticas simples, como no Código 3.4 ou políticas complexas, como no Código 3.5.

Código 3.4: Política simplificada usando *aCCountS-DSL*

```
1  Policy simples{
2     var {
3        utilizacao, total;
4     }
5     rules{
6        utilizacao = instance.tempoLigada * $tempoLigada;
7         total = utilizacao + instance.SomaSoftware
8     }
9     return total;
10 }
```

Na política *simples* são tarifados a quantidade de tempo em que a máquina virtual está ligada e os custos com software consumidos. Nas linhas 6-7 são definidas as regras da política, contabilizando a utilização da máquina virtual, multiplicando a quantidade de horas em que a mesma ficou ligada *instance.tempoLigada* pelo custo *$tempoLigada* de uma hora da máquina ligada. Por fim, a linha 7 calcula o total a pagar através da soma da utilização pelo custo com os software utilizados (*instance.SomaSoftware*). Na linha 9, o valor à pagar é retornado.

A política *sofisticada* busca exemplificar uma política mais complexa. As regras são definidas nas linha 8-35. Nas linhas 8, 9 e 10, as variáveis *descontoEnergia*, *descontoUsuario* e *taxaCentralDados* são inicializadas com 0, 0 e 0.14, respectivamente.

Código 3.5: Política mais sofisticada usando *aCCountS-DSL*

```
1  Policy sofisticada {
2    var {
3      taxaCentralDados, memoria, cpu, armazenamento;
4      upload,transacaoBD, custo, descontoEnergia;
5      descontoUsuario, desconto;
6    }
7    rules{
8      descontoEnergia = 0;
9      descontoUsuario = 0;
```

```
10      taxaCentralDados = 0.14;
11      memoria = instance.memoria * $memoria;
12      cpu = instance.cpu * $cpu;
13      armazenamento = instance.armazenamento * $armazenamento;
14      transacaoBD = instance.transacaoBD * $transacaoBD;
15      upload = instance.upload * $upload;
16      download = instance.download * $download;
17
18      if (instance.economiaEnergia >= 0.5 ){
19        descontoEnergia = 0.03;
20      }
21      if (instance.economiaEnergia >= 0.8 ){
22        descontoEnergia = 0.05;
23      }
24      if (instance.memoria >= 0.8 ){
25        descontoUsuario = 0.04;
26      }
27      if (instance.armazenamento >= 0.8 ){
28        descontoUsuario = 0.05;
29      }
30
31      custo = memoria + cpu + armazenamento + transacaoBD + upload + download
            + instance.software + taxaCentralDados + instance.servico;
32      desconto = instance.sla + descontoEnergia + descontoUsuario;
33      custo = custo - custo * desconto;
34    }
35    return custo;
36  }
```

Da linha 12 até a linha 15 são definidos os cálculos de uso dos recursos. Essa regra é definida pelo consumo em porcentagem de uso do recurso por hora e multiplicado pelo preço de 100% de sua utilização. Por exemplo: 0.50 de memória significa que foi consumido 50% de memória em uma hora e o preço é definido, também, em função da hora: $ 0.003 por 100% de uso de memória em uma hora. Para contabilizar o custo, multiplica-se o consumo do recurso por hora pelo preço em uma hora. Dessa forma o consumo de memória (linha 12), CPU (linha 13), armazenamento (linha 14) e transação no banco de dados (linha 15) foram calculados.

As linhas 16 e 17 calculam a quantidade de upload e download realizados na máquina virtual. O cálculo do consumo desses recursos é realizado pela multiplicação da quantidade de transações em uma hora e o preço da transação, como na linha 16, custo com os *uploads* e na linha 17, com os downloads.

Nas linhas 19, 22, 25 e 28 são definidos o comando de seleção *if*. Por esse comando, na linha 19 é verificado se a porcentagem de economia de energia é maior ou igual a 0.5 através do valor medido na máquina virtual pelo agente: *instance.economiaEnergia*, caso a condição seja verdadeira, a linha 20 é executada atribuindo à variável *descontoEnergia* o valor de 0.03 de desconto na cobrança. A condição da linha 22 é verificar se o valor de *instance.economiaEnergia* é maior ou igual a 0.8, caso seja verdadeiro a linha 23 é executada, atribuindo à variável *descontoEnergia* o valor de 0.05 de desconto na cobrança. A condição da

linha 25 verifica se o valor *instance.memoria* é maior ou igual a 80% de consumo de memória, se verdadeira, a linha 26 é executada atribuindo à variável *descontoUsuario* o valor de 0.04 de desconto na cobrança e a condição da linha 28 verifica se o valor *instance.armazenamento* é maior ou igual a 80% de consumo, se for verdadeira a sentença, a linha 29 é executada atribuindo à variável *descontoUsuario* o valor de 0.05 de desconto na cobrança.

Nas linhas 32 e 33 é atribuído à variável *custo* a soma dos custos pelo consumo dos recursos calculados, atribuídos às variáveis: *memoria*, *cpu*, *armazenamento*, *transacaoBD*, *upload* e *download*. Adiciona-se a esse total os gastos com software (*instance.software*), taxa de *datacenter* (*taxaCentralDados*) e serviços utilizados (*instance.servico*).

Na linha 34 é atribuída à variável *desconto* a soma dos descontos calculados no programa: *descontoEnergia* e *descontoUsuario* mais o valor de desconto por violação no SLA, por meio do valor medido pelo agente: *instance.sla*. Na linha 35 é calculado o custo a ser pago, calculando os descontos recebidos. E por fim, na linha 37 o valor do custo é retornado da função e será utilizado para cobrar o cliente.

Assim como políticas simples ou sofisticadas, os administradores da nuvem podem criar políticas que visam o bem estar social, como no código 3.5, no qual o administrador criou regras para o incentivo à economia de energia, por meio de descontos quando o cliente assim o fizer (linhas 19-24). De forma semelhante, descontos podem ser ofertados para clientes que utilizam muitos recursos da máquina virtual, como memória e armazenamento (linhas 25-30). Por fim, em caso de violação de SLAs, descontos para o cliente podem ser acordados (linha 34).

Quando o administrador da nuvem define as políticas de tarifação, um cliente pode escolher uma política para ser tarifado e entender todos os critérios de cobrança por meio das regras definidas para aquela política, lhe tornando claro o modo como está sendo tarifado pelo serviço.

Após a configuração realizada (políticas definidas por meio da *aCCountS-DSL*, preços e recursos determinados no perfil das máquinas), a execução do serviço pode ser iniciada para o *aCCountS* realizar a tarifação das nuvens que a utiliza. Com a finalidade de validar a arquitetura, o serviço e a linguagem, testes foram feitos no *aCCountS* e no *aCCountS-DSL* a fim de realizar uma avaliação experimental, que está detalhada no próximo capítulo.

3.1.5 Conclusão

Este capítulo apresentou a proposta desse trabalho de mestrado, detalhando o serviço proposto *aCCountS*, por meio da sua arquitetura e a linguagem proposta, a *aCCountS-DSL*. Seu objetivo foi flexibilizar a tarifação na computação em nuvem. Isso foi feito por meio de um serviço suficientemente simples (fácil de utilizar) e flexível (definição de regras baseadas em vários requisitos), para atender diferentes formas de tarifação de serviços em diferentes nuvens de infraestrutura. Isso, por meio da linguagem *aCCountS-DSL*, em que diferentes formas de cobrança podem ser definidas, atendendo tanto as necessidades da academia por uma precificação mais abrangente, como o modelo utilizado pelo mercado.

Algumas funcionalidade não foram implementadas, ou por questão de tempo ou por questão de escopo. A funcionalidade que atende ao requisito de tarifação, modelos (pré-pago e pós-pago) não foi concluída, por limitação do tempo, por exemplo. Por outro lado, questões como a automatização na criação de agentes a partir do serviço de tarifação e o atendimento aos requisitos de segurança e disponibilidade, importantes em serviço na nuvem, não foram implementados, pois estavam fora do escopo do trabalho.

4 O PROTÓTIPO DO *ACCOUNTS*

Neste capítulo será apresentado o protótipo que foi implementado baseado nas ideias expostas no capitulo anterior, com suas principais características e as escolhas de projeto efetuadas durante sua concepção. Esse protótipo serviu de base para a realização da avaliação experimental.

4.1 Arquitetura completa do protótipo

A Figura 4.1 apresenta a arquitetura dos protótipos implementados. Nela pode-se perceber que os dois elementos, serviço e agente, foram implementados utilizando o *framework Ruby on Rails*.

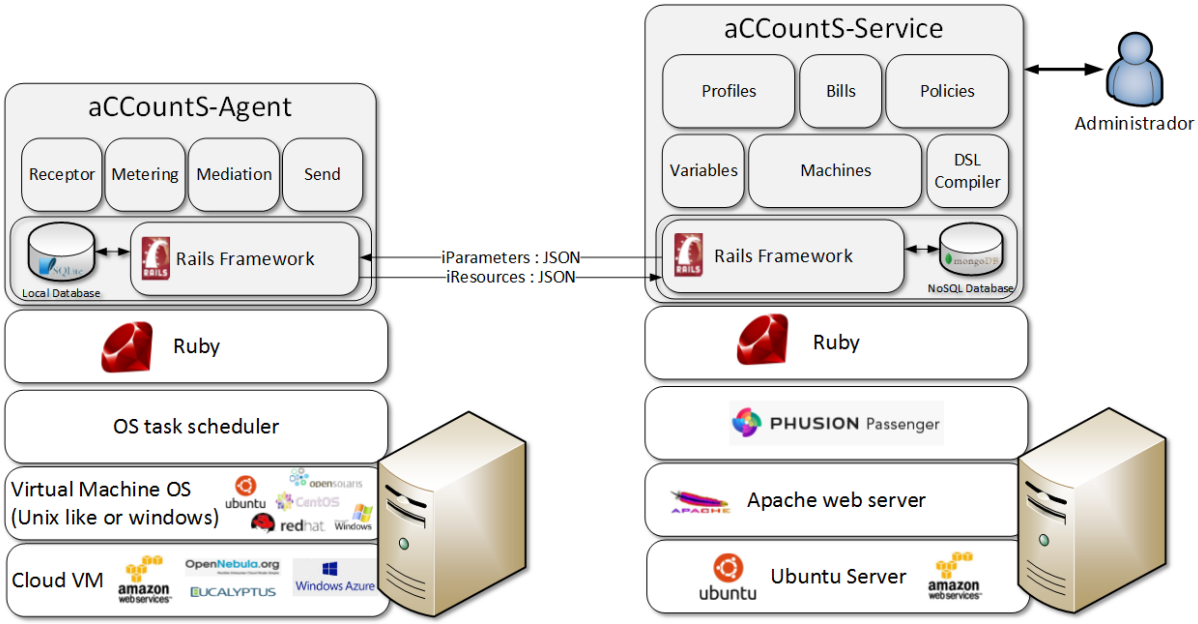

Figura 4.1: Arquitetura do protótipo do *aCCountS*

Do lado do agente, percebe-se que sua implantação se dará em uma nuvem de infraestrutura qualquer, na camada mais inferior. Logo acima, temos o nível da máquina virtual, que pode ser implantada com qualquer sistema operacional *Unix like* (diversas versões de *linux*) ou *Windows*. Na máquina virtual, a execução das tarefas do agente é baseada no agendador de tarefas, responsável por chamar, de tempos em tempos, os componentes do agente sem sobrecarregar o processamento da própria máquina virtual. Logo acima, tem-se a plataforma *Ruby*, necessária para executar os códigos nessa linguagem, dando suporte à efetiva implementação do agente. Por fim, os componentes do agente (*Receptor, Metering, Mediation e Send*) foram implementados conforme o *framework Ruby on Rails* e usam uma base de dados local *SQLite* para armazenamento de informações.

A implementação do serviço possui, como camada mais baixa uma plataforma servidora que disponibiliza o serviço para comunicação. Nesse caso, foi escolhido um servidor na

Amazon, utilizando o sistema operacional *Ubuntu Server*. O servidor *web Apache* foi instalado e configurado para implantação do serviço. Sobre o *Apache*, foi necessária a instalação do *Phusion Passenger*, componente responsável por integrar a linguagem *Ruby* no servidor *Apache* e, assim, tornar possível a execução do serviço, feito em *Ruby on Rails*. Logo acima, assim como no agente, tem-se a linguagem *Ruby*, também necessária para executar a aplicação, assim como no agente. Por fim, a implementação do serviço está sobre essa camada, onde o serviço utilizou o *framework Ruby on Rails* e o banco de dados *NoSQL mongoDB* para armazenamento de dados, de forma a prover acesso e execução das funcionalidades relacionadas a variáveis, perfis, máquinas, contas políticas e o compilador da aCCountS-DSL (*DSL Compiler*). O administrador interage diretamente com as camadas superiores do aCCountS-Service.

Para comunicação entre o agente e o serviço, a Figura 4.1 mostra as duas interfaces, *iParameters* e *iResources*, cujo conteúdo transmitido é no formato *JSON* e são responsáveis por levar as variáveis de monitoramento do serviço ao agente e levar os dados de uso do agente ao serviço.

4.2 Implementação da aCCountS-DSL

O primeiro passo para implementação da proposta foi a criação da aCCountS-DSL, através da qual cada máquina pode ser tarifada de forma diferente. Criar uma DSL significa implementar um compilador (ou interpretador) de uma linguagem e, por isso, tem-se várias alternativas de implementação, desde o uso de *frameworks* que automatizem esse processo, até a implementação manual de todos os componentes do compilador (analisador léxico, analisador sintático, gerador de código).

Nesse cenário, o Xtext (XTEXT, 2013) surge como uma alternativa viável no sentido de facilitar a criação da linguagem e posterior geração do compilador. Xtext é um *framework* para desenvolvimento de linguagens de programação e linguagens de domínio específico. Ele cobre todos os aspectos de uma infraestrutura completa de linguagem, como analisadores léxico e sintático e possui integração com a IDE Eclipse.

Utilizando esse *framework* foi possível criar a gramática da linguagem pretendida e, a partir dela, gerar analisadores léxico e sintático que foram usados como base para o compilador. Também foi possível testar as definições da linguagem de forma simples, usando a integração ao Eclipse na forma de um editor contendo a linguagem definida. No Apêndice A, tem-se a gramática desenvolvida com as definições suportadas pela linguagem.

4.3 O Compilador da DSL (*DSLCompiler*)

A partir das regras definidas na gramática, o Xtext é capaz de gerar analisadores léxico e sintático, no entanto, para processar o código recebido no formado da DSL é necessário implementar um gerador de código que possa transpor o resultado das análises para um código que possa ser utilizado em uma aplicação.

Para atingir esse objetivo, foi utilizada a biblioteca Xtend (XTEND, 2013), a qual se integra ao Xtext e torna possível a implementação de um gerador de código de maneira facilitada. Essa biblioteca, por padrão, torna possível a transformação do código da DSL em código Java. Pode-se, ainda, exportar o compilador no formato *jar* (*Java ARchive*, um "executável" *Java*) para utilizá-lo de forma independente da IDE Eclipse, recebendo arquivos de código-fonte da DSL como entrada e devolvendo arquivos fonte *Java* como saída da compilação.

Por motivos que serão explicados na seção 4.4, o protótipo do serviço foi implementado na linguagem de programação *Ruby* (RUBY, 2013), usando o *framework Rails*. Então, para total integração dos códigos gerados pelo compilador com a aplicação feita, o gerador de código foi feito de forma a gerar código Ruby, e não Java. Além disso, diversas outras vantagens foram obtidas com essa estratégia, como a possibilidade de executar o código gerado na aplicação alvo; classes abertas (característica da linguagem *Ruby*) facilitam a implementação da herança com execução das regras da classe pai e da classe filha e independência da plataforma Java para usar a linguagem (o que seria um *overhead* a mais se fosse preciso usar as duas linguagens, *Ruby* e *Java*).

No Código 4.1 é possível visualizar um trecho do gerador de código implementado com Xtend e interpretando a definição básica de uma política e gerando uma classe *Ruby* equivalente. No apêndice B pode-se ver todo o gerador implementado. As marcações entre "<<" e ">>" são responsáveis por testar e iterar os *tokens* vindos da DSL e, através deles, efetuar testes e gerar código *Ruby* equivalente.

Código 4.1: Trecho do gerador de código em Xtend

```
1  def compile(Policy p) '''
2    class <<p.name.toFirstUpper>><<IF p.supertype!= null>> < <<p.supertype>>
       <<ENDIF>>
3      def self.execute(machine, instances = {})
4        <<IF p.supertype!= null>>super<<ENDIF>>
5        parameters = {}
6        prices = []
7        <<FOR ps:p.eAllContents.toIterable.filter(typeof(Price))>>
8          prices << "<<ps.name>>"
9        <<ENDFOR>>
10       parameters = JSON.parse(machine.status(prices))
11       parameters["instances"] = instances
12       <<FOR f:p.rules>>
13         <<f.compile>>
14       <<ENDFOR>>
15       <<p.returning.compile>>
16     end
17
18     <<FOR vars:p.variable>>
19       attr_accessor :<<vars.name>>
20     <<ENDFOR>>
21   end
22 '''
```

4.4 O protótipo do serviço aCCountS

Feito o compilador, o próximo passo foi criar um protótipo do serviço para integrá-lo e poder testar seu funcionamento. Desse modo, foi projetado um protótipo como uma aplicação *web*, utilizando a linguagem *Ruby* e o *framework Rails*. A escolha dessa linguagem se deu pelo possibilidade de disponibilizar de forma simples interfaces de comunicação do agente com o serviço no formato de *web services*. Assim, a mesma implementação torna possível o acesso à interface *web* pelo administrador do serviço e pelos agentes espalhados nas máquinas virtuais usando a troca de mensagens no formato JSON com o serviço *web*. Isso fez com que fosse possível integrar o compilador à aplicação e executar os códigos resultantes como códigos nativos, por se tratar, também, de código *Ruby*. O protótipo pode ser acessado no endereço http://accounts.zn.inf.br.

Pode-se citar outras decisões de projetos que foram tomadas para a implementação do serviço como, por exemplo, o uso de uma base de dados *NoSQL*, o MongoDB (MONGODB, 2013), para a persistência de dados do aCCountS. Nesse caso, essa escolha se deveu pela grande quantidade de dados que precisaria ser armazenada e processada para cada máquina virtual cadastrada. Assim, obteve-se vantagens como alta performance, não utilização de esquemas (possibilitou com que itens da mesma entidade pudessem ter atributos diferentes, pois cada máquina pode utilizar variáveis diferentes na sua medição), orientação a documentos (o MongoDB usa o formato JSON, o que trouxe facilidades formatação das mensagens a serem trocadas com o agente), além de se permitir o aninhamento de dados em complexas hierarquias (as informações persistidas incluíram as políticas, as relações de herança e os dados de uso das máquinas).

O protótipo do agente também foi implementado seguindo as mesmas ideias; no entanto, devido a necessidade de ser mais simples para não comprometer o desempenho da máquina monitorada, outras decisões de projeto foram tomadas. A primeira é que, apesar de implementado como aplicação *web*, o agente não é executado dessa forma, pois necessitaria de um servidor *web* instalado e configurado para isso. Ao invés disso, as tarefas específicas para seu funcionamento são chamadas diretamente através do agendador de tarefas do sistema operacional. Entre essas tarefas se encontram a requisição ao serviço dos recursos a serem medidas, a medição desses recursos, a mediação dos mesmos e, por fim, o envio dos dados processados ao serviço.

Da mesma forma, a persistência dos dados foi trocada por um formato mais simples. Como o agente só irá armazenar dados de uma máquina, sua necessidade é bem menor que a do serviço. Dessa forma, escolheu-se o banco de dados SQLite (SQLITE, 2013), uma biblioteca escrita na linguagem C, que faz o papel de banco de dados de forma autocontida e sem necessidade de servidor.

Devido ao uso do *framework* Ruby on Rails, as principais ideias implantadas por ele foram seguidas no desenvolvimento do protótipo, como: (i) DRY (*don't repeat yourself* - não se repita), que é o conceito da ideia de se definir nomes, propriedades e códigos em somente um lugar e reaproveitar essas informações em outros, o que estimula a reutilização de código entre componentes, e (ii) *Convention over Configuration* (Convenções sobre Configurações),

que diz que deve-se assumir valores padrão onde existe uma convenção (se o desenvolvedor quiser, pode sobrescrever essa convenção com o valor necessário, por exemplo, uma classe *User* pode ter seus dados armazenados na tabela *Customer*. Seguindo a convenção, seria na tabela *Users*) e com isso, o tempo de desenvolvimento cai ainda mais pois precisa-se apenas seguir as recomendações do *framework* (como nomes de arquivos, localização de pastas, etc.) para a aplicação funcionar.

Da mesma forma, alguns padrões de projeto foram adotados durante o desenvolvimento, tanto por o *framework* influenciar nesse uso, quanto por decisões de projeto para o funcionamento da arquitetura do protótipo. São eles: (i) o padrão MVC (model-view-controller), usado na estruturação das camadas da aplicação, tanto no serviço quando no agente; (ii) REST (*Representational State Transfer*), usado no serviço para a disponibilização de recursos através de rotas para acesso do agente; (iii) Singleton, principalmente no agente, que deve ter referência única para o recebimento de informações, monitoramento da máquina e envio de dados para o serviço; (iv) Template Method, tendo seu uso nas classes geradas pelo compilador da DSL, de forma a ter-se métodos genéricos que se adaptam de acordo com o código gerado e (v) Observer, acoplado ao padrão MVC e, assim, utilizado na arquitetura do sistema para disparar métodos em caso de mudança de estado em objetos.

Na próxima subseção são mostradas com mais detalhes as principais funcionalidades implementadas no protótipo do *aCCountS*.

4.4.1 Tela inicial do serviço aCCountS

A Figura 4.2 é a tela inicial para o administrador do sistema, que dá acesso ao controle de usuários. Na parte superior da tela está disposto o menu com as opções de gerenciamento: *Variables*, cadastro das variáveis a serem monitoradas no agente, *Profile*, cadastro dos perfis de máquinas disponíveis na nuvem e os respectivos preços dos seus recursos, *Policies*, criação das políticas de tarifação e *Machine* configuração da forma de cobrança da máquina virtual, por meio da combinação de uma política de tarifação e um perfil de máquina. Na tela são mostrados os demais usuários do sistema com a opções de excluir usuário ou editá-lo.

4.4.2 Cadastro de variáveis (recursos a serem medidos)

A Figura 4.3 é a tela *Variables*. A partir dela, são criadas as variáveis que representarão os recursos a serem medidos. Esses recursos são definidos com um nome para identificá-lo, uma descrição opcional e os comandos responsáveis por monitorá-los, nos sistemas operacionais *Windows* ou *Linux*. Na figura estão cadastradas as variáveis cpu, memoria, armazenamento e armazenamento, como as duas últimas variáveis representam o mesmo recurso, há uma descrição indicando que a primeira será usada para máquinas virtuais com sistema operacional *Linux* na nuvem da Amazon e a segunda, será usada para máquinas virtuais, também com *Linux*, mas na nuvem *Windows Azure*. Isso precisa ser feito por haver peculiaridades de configuração nas diferentes nuvens que precisam ser abordadas no monitoramento dos recursos. Cara recurso criado no sistema pode ser alterado ou excluído por meio dos botões: *Edit* e *Delete*, respectiva-

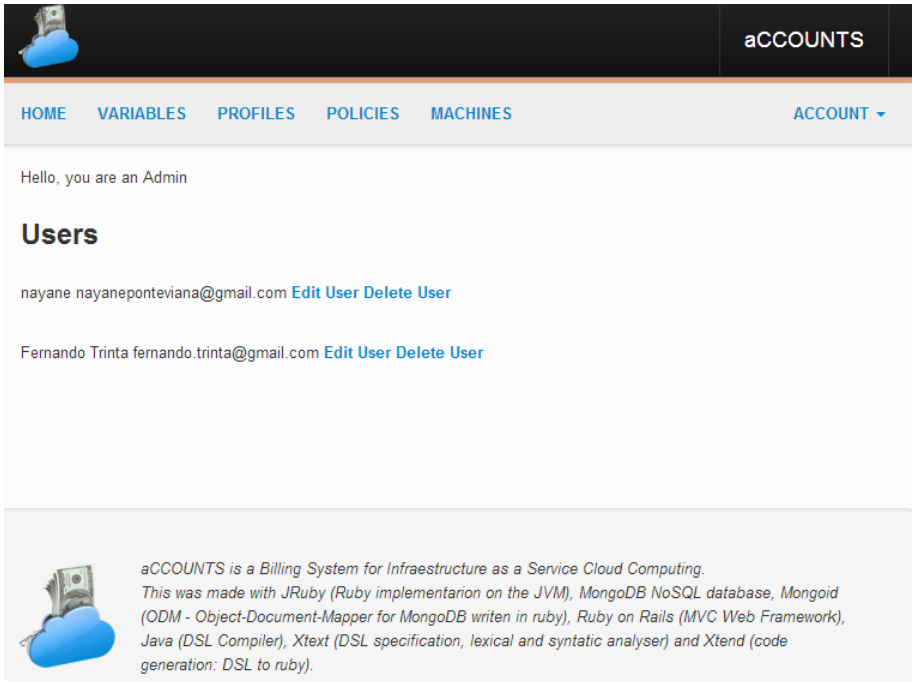

Figura 4.2: Protótipo da arquitetura *aCCountS* - Tela inicial

mente.

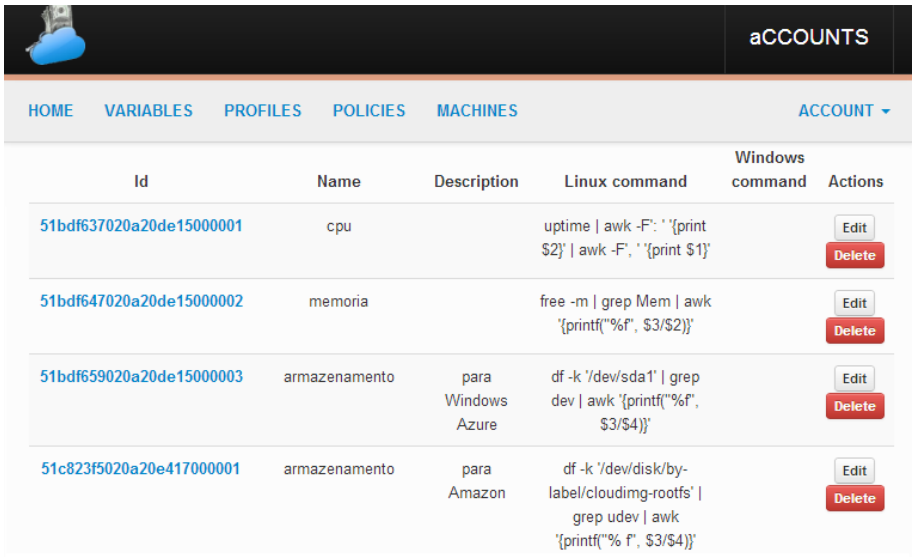

Figura 4.3: Protótipo da arquitetura *aCCountS* - Tela de variáveis

A Figura 4.4, a seguir, mostra a tela responsável pelo cadastro de uma nova variável. Nela, pode-se ver o formulário com os campos necessários de serem preenchidos para que a variável seja cadastrada (*Name*, *Description*, *Linux command* e *Windows command*).

4.4.3 Cadastro de Perfis

A Figura 4.5 ilustra a tela de lista de perfis. Cada perfil possui um nome, uma descrição opcional, o usuário responsável por sua criação e um conjunto de variáveis relacio-

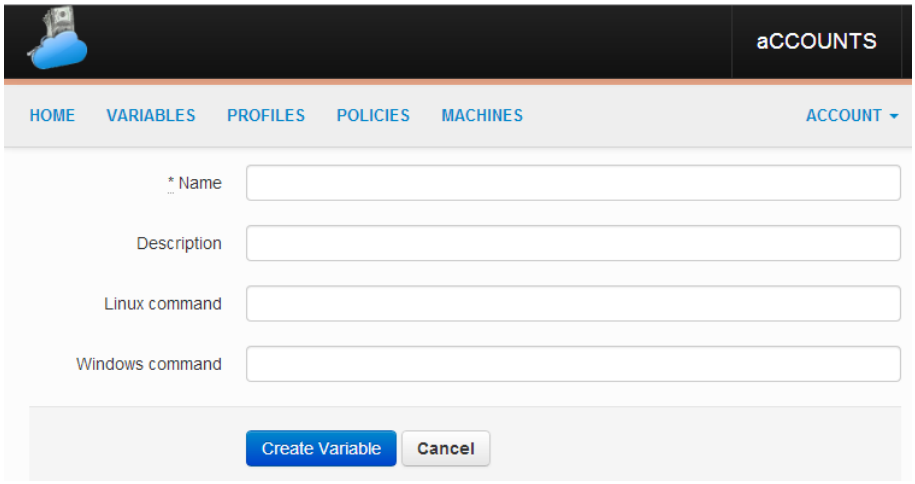

Figura 4.4: Protótipo da arquitetura *aCCountS* - Cadastro de variável

nadas, cada uma com um preço específico. São esses valores cadastrados que serão usados no momento da tarifação (execução da política) em substituição ao respectivo comando. Os perfis podem ser criados pelo administrador, como também podem ser editados e excluídos nos botões correspondentes.

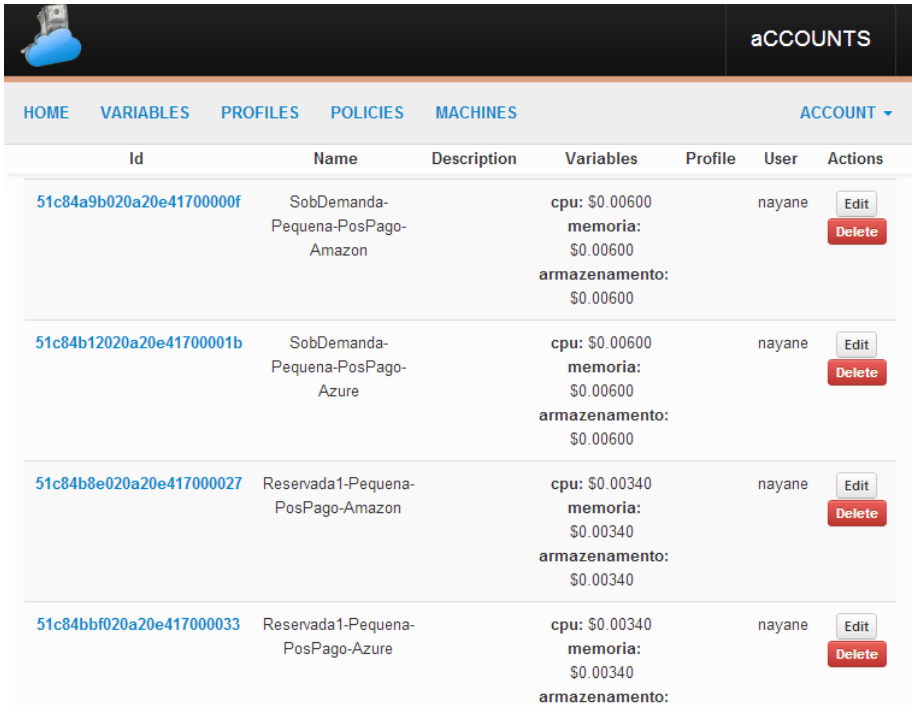

Figura 4.5: Protótipo da arquitetura *aCCountS* - Tela de perfis

A Figura 4.6 mostra a tela de criação de um perfil. Nesse cadastro o administrador define um nome para o perfil, pode inserir alguma descrição e as informações sobre os recursos de hardware disponíveis, como memória, CPU, armazenamento, download e upload para facilitar a identificação do perfil com as máquinas virtuais configuradas na nuvem. Por fim o administrador adiciona os recursos que são disponibilizados para aquele perfil, podendo selecionar qualquer quantidade de recursos e tendo que inserir o preço de cada um deles.

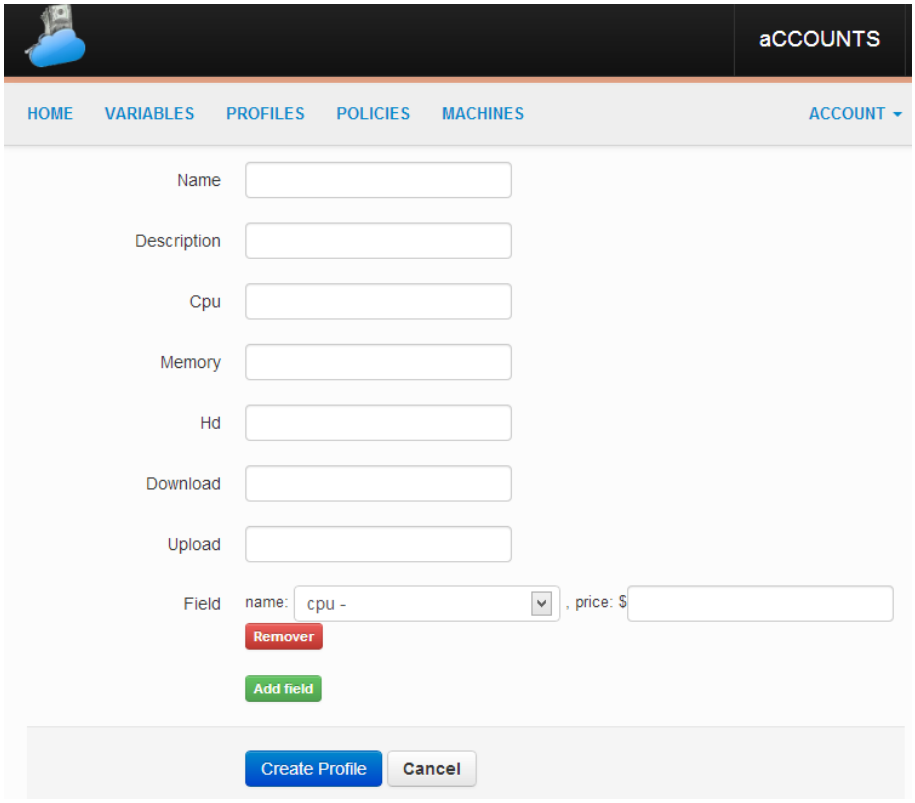

Figura 4.6: Protótipo da arquitetura *aCCountS* - Criar perfil

4.4.4 Cadastro de Políticas

Na Figura 4.7 é ilustrada a tela correspondente ao menu *Policies* em que são listadas as políticas cadastradas. Cada política tem, basicamente, um nome e um conteúdo, que são as regras definidas por meio da linguagem *aCCountS-DSL*. Para criar uma política, no entanto, o administrador somente precisa digitar o conteúdo da política, escrito na *aCCountS-DSL*, para que o sistema invoque o *DSLCompiler* e verifique se a política está correta. Passando nesse teste, a política pode ser cadastrada e outros dados vão ser inferidos a partir desse conteúdo, como o nome da política e uma possível classe mãe, se houver herança. Cada política cadastrada pode ser editada ou excluída pelo administrador e novas políticas podem ser criadas utilizando os respectivos botões, similares aos das outras telas, mas que não aparecem na Figura 4.7.

A Figura 4.8 mostra a tela de cadastro de uma política. Conforme dito anteriormente, para o cadastro de uma política é necessário apenas o código escrito na *aCCountS-DSL*, por isso, a tela de cadastro possui apenas um campo para que o código possa ser digitado. Caso não haja erros de sintaxe ou de semântica, a política pode ser compilada e o código *Ruby* correspondente será gerado. Para facilitar a entrada do código da política, no campo de inserção da política o texto da mesma é destacado conforme a sintaxe definida.

Caso haja erros na compilação da política, esses erros que forem encontrados pelo *DSLCompiler* são repassados à aplicação e mostrados na mesma tela de cadastro para que o administrador possa corrigi-los, como pode ser visto na Figura 4.9. Nesse exemplo, a palavra reservada *extends* está grafada de maneira errada, então o sistema mostra esse erro, vindo do

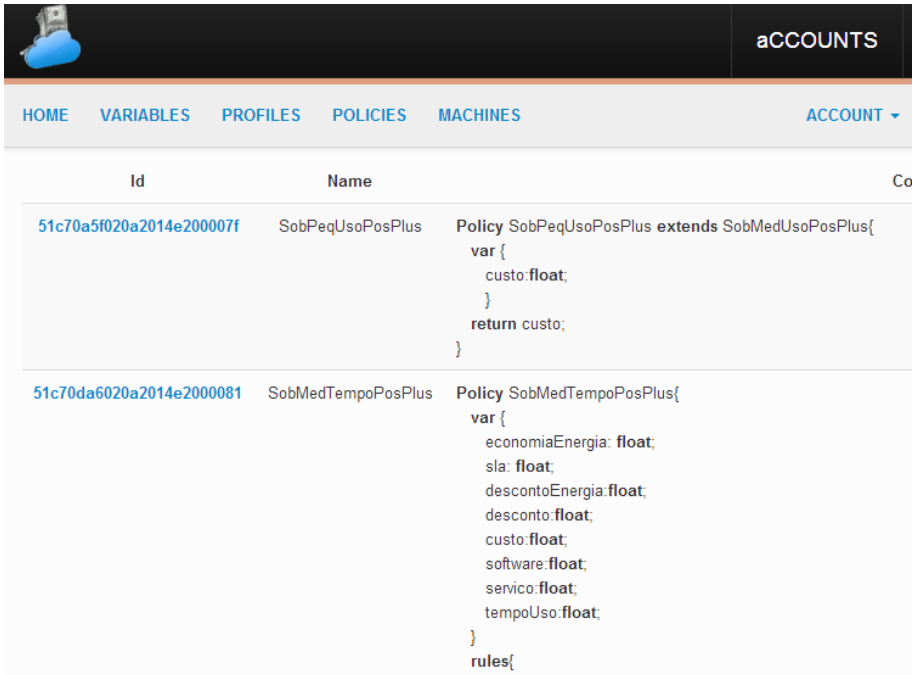

Figura 4.7: Protótipo da arquitetura *aCCountS* - Tela de políticas

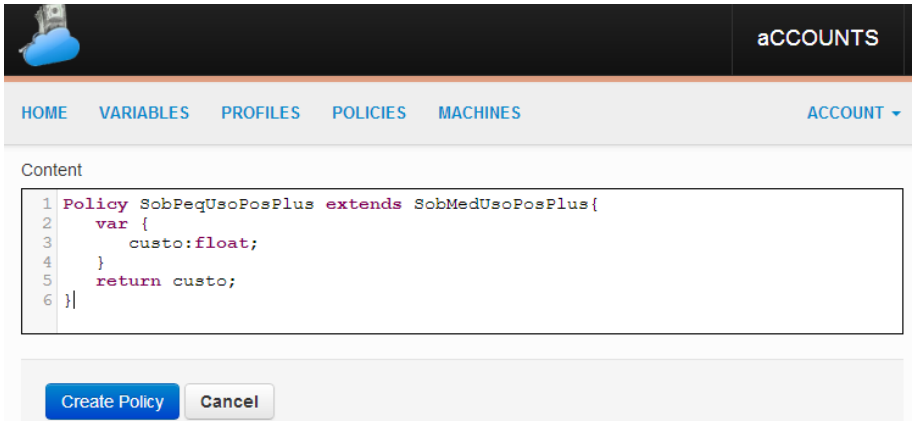

Figura 4.8: Protótipo da arquitetura *aCCountS* - Cadastro de política

DSLCompiler.

Na tela de visualização de uma política é possível verificar o código *Ruby* que foi gerado na compilação da política. Essa tela é aberta logo após o cadastro da politica para que o administrador veja se correu tudo bem na criação da política. A Figura 4.10 mostra um exemplo dessa tela para a política mostrada na Figura 4.10, onde pode-se ver indicado o nome da política, o nome da política a qual ela herda, o código dela escrito na *aCCounts-DSL* e o código *Ruby* gerado pela compilação.

Para que a estratégia de herança de políticas funcione corretamente, a solução adotada foi adicionar um método na politica que retorne seu conteúdo compilado juntamente com o conteúdo da política a qual herda. Dessa forma, pode-se capturar as definições, em qualquer nível de aninhamento, da políticas superiores e, ao executar, o código precisa estar junto para o interpretador *Ruby* poder reconhecer as definições. A figura 4.11 mostra um trecho de código

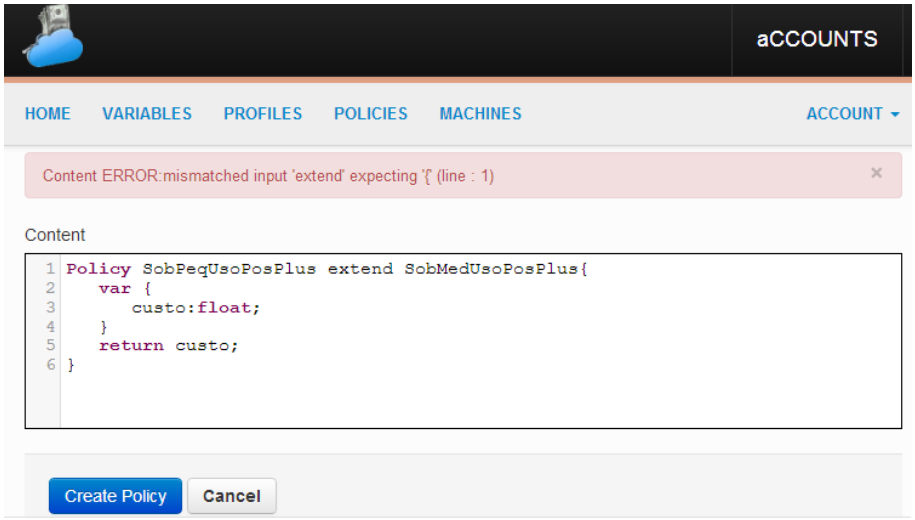

Figura 4.9: Protótipo da arquitetura *aCCountS* - Política com erros

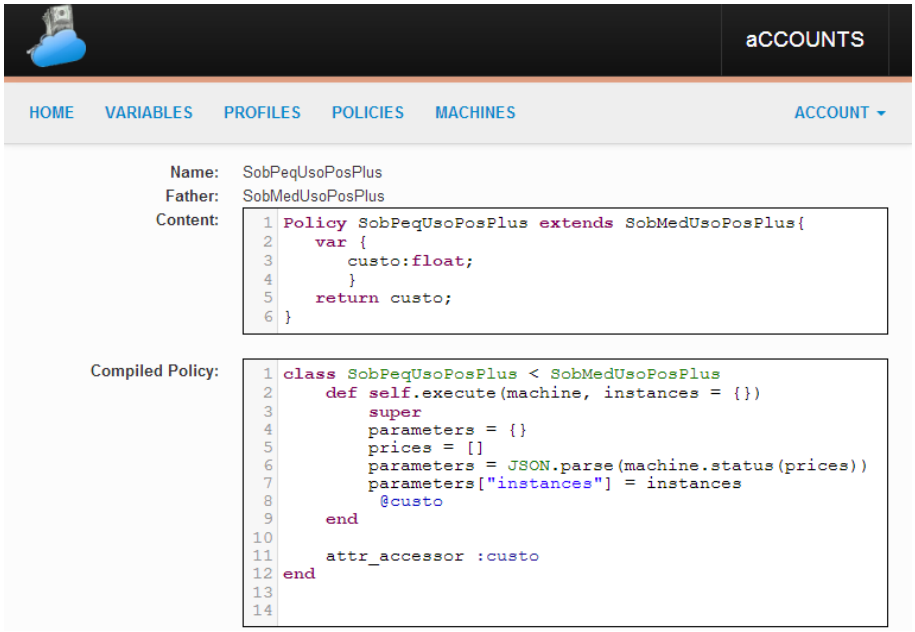

Figura 4.10: Protótipo da arquitetura *aCCountS* - Visualizar política

completo gerado dessa junção.

4.4.5 Cadastro de Máquinas virtuais

A Figura 4.12 ilustra a funcionalidade de manipulação das máquinas ou instâncias do *aCCountS-Service*. No caso dessa funcionalidade, o administrador da provedora de nuvens pode cadastrar instâncias, e associá-las a uma política de tarifação e um perfil criados anteriormente, e disponibilizá-las aos clientes. O mesmo escolhe, então, uma para ser utilizada na tarifação de sua máquina virtual em que o perfil de hardware corresponda a suas necessidades.

Com o sistema configurado, o serviço está apto a enviar as variáveis para serem medidas pelo agente assim que o mesmo solicitar e posteriormente receber os registros de me-

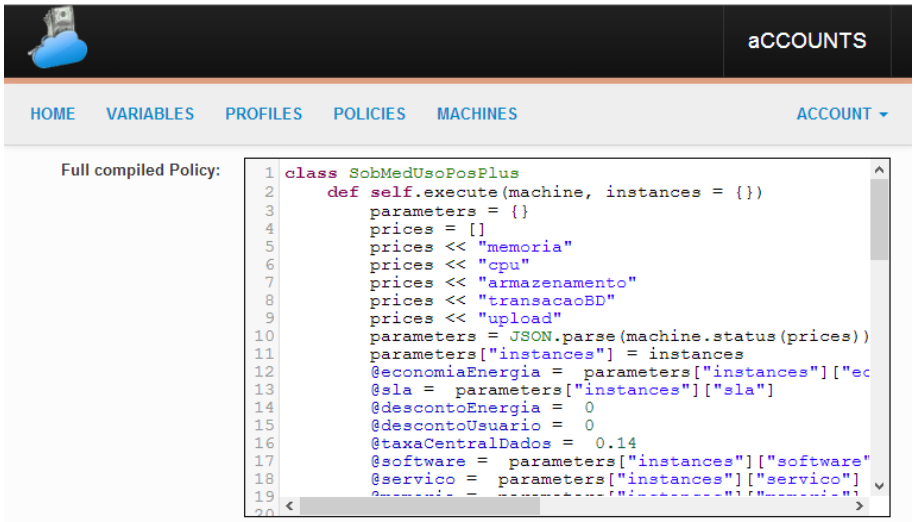

Figura 4.11: Protótipo da arquitetura *aCCountS* - Trecho Código *Ruby* completo com códigos herdados

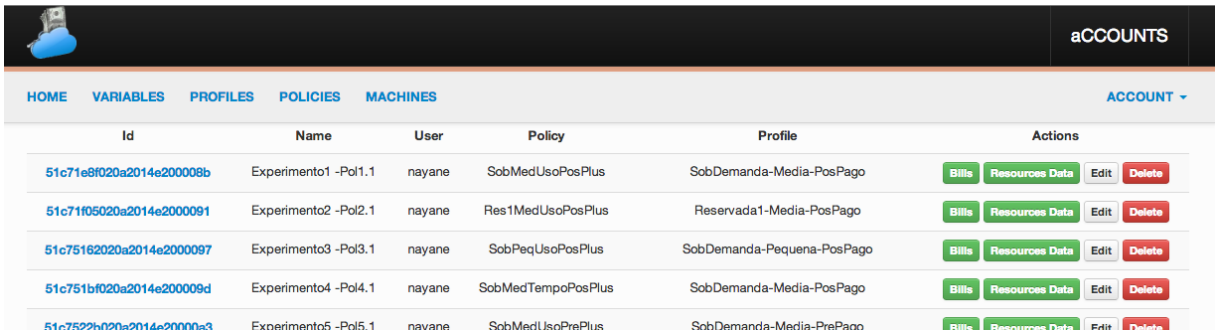

Figura 4.12: Protótipo da arquitetura *aCCountS* - associação entre clientes, perfis e políticas

dição, que ativam a função de precificação e a cobrança é calculada para cada registro como ilustra na Figura 4.13. Nessa tela são listados os registros de medição, o momento da medição, os recursos medidos e seus respectivos valores, através do formato de arquivo JSON, o valor da cobrança e a condição se o registro foi processado pela função de tarifação e enviado ao cliente (True) ou não (False). Além disso, o administrador pode editar e excluir qualquer registro.

A cada hora os dados de registros são recebidos e o valor da cobrança é calculado e registrado na tela *Resources Data*. Tanto os dados dos recursos e seus valores de cobrança, *Resources Data*, como o valores da fatura são obtidos por máquina virtual, por meio da tela *Machine*, onde o administrados pode acionar o botão *Resources Data* para acompanhar os registros, como acionar o botal *Bill* para calcular a tarifação. Na figura 4.14 é possível visualizar registros de tarifação recebidos para uma maquina específica. Esses registros possuem dados mediados pelo agente no período correspondente a um hora.

4.4.6 Envio de dados ao agente

Para que o serviço envie as informações que o agente precisa para efetuar seu processamento, é necessário um processo que selecione, de acordo com a maquina, as variáveis

Figura 4.13: Protótipo da arquitetura *aCCountS* - Registros de cobrança

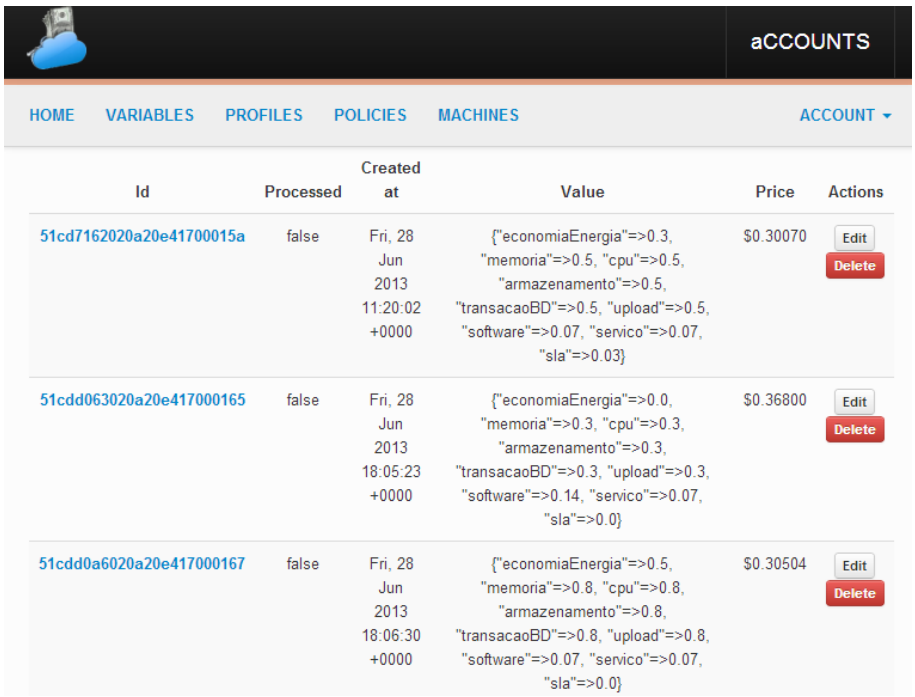

Figura 4.14: Protótipo da arquitetura *aCCountS* - dados de uso recebidos

cadastradas para monitorá-la. Isso é feito segundo o Código 4.2.

Código 4.2: Protótipo da arquitetura - Envio de variáveis para o agente

```
1 def get_parameters
2   @machine = Machine.find(params[:id])
3   render json: @machine, only: :profile, include: { profile:
4     { only: :profile_variables, include: { profile_variables:
5       { only: [:variable], include: { variable: { only: [:name, :
            linux_command, :windows_command] } } } }
6     }
7   }
8 end
```

Nesse código, na linha 2, a máquina correspondente ao identificador que é passado por parâmetro é selecionada no banco de dados. A partir disso é montada a lista com as variáveis correspondente ao perfil da máquina selecionada, enviando somente os nomes dos recursos e os comandos para medi-las (linhas 3 a 7).

4.4.7 Recebimento de dados do agente

Da mesma forma, é necessário que o serviço seja capaz de receber os dados de uso enviados pela agente na máquina virtual e os processe para calcular o valor de custo, de acordo com a política estabelecida. O Código 4.3 faz esse papel.

Código 4.3: Protótipo da arquitetura - Recebimento de dados do agente

```
1 def set_resources
2   @machine = Machine.find params[:machine_id]
3   params[:_json].each do |p|
4     @resource_data = @machine.resources_data.build(p)
5     @resource_data.save
6   end
7   render text: params.to_json
8 end
```

Ao ser chamado, esse método deve receber o identificador da máquina para carregá-la do banco de dados, assim como o método anterior. Com a máquina selecionada, são inseridos os registros de medição recebidos do agente (linhas 3 a 6) e são criados os registros de uso (*resource_data*). No momento da criação de cada registro de uso, é aplicado o código da política com a função *pricing*, chamada na linha 5 (ao salvar) e que pode ser vista no Código 4.4. Por fim, uma resposta é enviada (linha 7) para garantir ao agente que os dados foram processados.

Código 4.4: Protótipo da arquitetura - Aplicação da política a registro de medição

```
1 def pricing
2   parameters = self.attributes
3   eval self.machine.policy.full_compiled_policy
4   self.price = eval(self.machine.policy.name).execute(@machine, parameters)
5 end
```

A função *pricing* faz a avaliação do código da política completo (política + políticas das quais herda) na linha 3. Após isso, o preço do recurso é atribuído na linha 4 como sendo a execução da política (método *execute*) recebendo como parâmetros a máquina e os parâmetros do registro de medição.

4.5 O protótipo do aCCountS-Agent

Conforme mencionado na seção anterior, o protótipo do aCCounts-Agent também foi implementado usando o *framework Ruby on Rails*, mas, nesse caso, trata-se de uma aplicação mais simples, com um pequeno número de métodos a serem executados para realizar as tarefas de medição e mediação. A seguir, esses métodos serão explicados com mais detalhes.

4.5.1 Coleta de recursos a serem medidos

O primeiro processo apresentado é o de coleta de variáveis para que a máquina virtual possa ser monitorada. O Código 4.5 é o método que faz essa tarefa.

Código 4.5: Protótipo do agente - Método de coleta de recursos a serem medidos

```
1  def self.get_rules
2    require 'net/http'
3    url = "#{ENV['SERVER_URL']}/machines/#{ENV['MACHINE_ID']}/i_parameters"
4    data = JSON::parse Net::HTTP.get(URI.parse(url))
5    Rule.inactivate_all
6    data['profile']['profile_variables'].each do |v|
7      command = v["variable"]["#{ENV['HOST_OS']}_command"]
8      if r = Rule.find_by_name(v['variable']['name'])
9        r.update_attributes(command: command, active: true)
10     else
11       Rule.create(name: v['variable']['name'], command: command)
12     end
13   end
14 end
```

Para que seja feito com sucesso, é necessário configurar algumas variáveis, a saber: (i) o endereço do servidor aCCountS, (ii) a identificação da máquina e (iii) o sistema operacional da máquina virtual. Esses dados são representados nas linhas 3 e 7, com as variáveis globais *ENV*[*'SERVER_URL'*], *ENV*[*'MACHINE_ID'*] e *ENV*[*'HOST_OS'*].

De posse dessas informações, primeiramente é montado o endereço completo para o qual serão requisitadas as variáveis. Na linha 3 tem-se a junção do endereço do servidor com a identificação da máquina seguido do caminho do *i_parameters*, que é a interface que devolve as variáveis correspondentes à máquina. O resultado dessa requisição é transformado no formato *JSON* (já que na requisição os dados são efetivamente transferidos como texto) na linha 4.

A linha 5 desse código desativa todas as regras previamente armazenadas no banco de dados, para, no caso das regras terem sido alteradas no serviço *aCCountS*, o agente ficar com as definições mais novas. Entre as linhas 6 e 13, as regras que foram lidas são percorridas e, para cada uma delas, o comando será armazenado na tabela de regras do agente de acordo com o sistema operacional correspondente. Se já existir uma regra armazenada para aquela variável no banco, ela será atualizada, caso contrário, será criada uma nova.

4.5.2 Medição

O processo de mediação utiliza as regras armazenadas pelo processo anterior e as executa no sistema operacional da máquina virtual segundo o método mostrado no Código 4.6.

Código 4.6: Protótipo do agente - Medição de dados de uso

```
1  def self.metering
2    rules = Rule.where(active: true)
3    if rules.empty?
```

```
4       self.get_rules
5     else
6       rules.each do |rule|
7         r = `#{rule.command}`
8         ResourceRawData.create(value: r.to_f, rule: rule)
9       end
10    end
11  end
```

Nesse código, todas as regras que estão atualmente ativas (na última vez que foram trazidas do serviço *aCCountS*) são trazidas do banco, na linha 2, para serem percorridas. Se não existirem regras ainda no banco, será executado o método anterior, para efetivamente trazer essas regras antes de executar a medição (linha 4).

Caso existam regras, elas serão percorridas e executadas, uma a uma, na repetição das linhas 6 a 9. O comando é executado na linha 7 e o resultado dessa execução é armazenado como um *ResourceRawData* (recurso não processado) na linha 8.

4.5.3 Mediação

Após a medição de várias vezes dos recursos da máquina, de tempos em tempos, o agente calcula a média desses recursos utilizados, de forma a pré-processar essas informações e demandar menos recursos na hora de enviar esses dados ao serviço ao *aCCountS*, conforme ilustrado no Código 4.7.

Código 4.7: Protótipo do agente - Mediação nos dados de uso coletados

```
1  def self.mediating
2    rules = Rule.where(active: true)
3    if rules.empty?
4      self.get_rules
5    else
6      rules.each do |rule|
7        media = rule.resources_raw_data.where(processed: false).average(:
             value)
8        ResourceData.create(value: media.to_f, rule: rule)
9        rule.resources_raw_data.where(processed: false).update_all(processed:
              true)
10     end
11   end
12 end
```

O processo de mediação começa da mesma forma que o de medição. Primeiramente ele tenta trazer as regras de medição da máquina (linha 2), caso não existam faz novamente o processo de requisição delas ao serviço *aCCountS*.

Se existirem as regras, entre as linhas 6 e 10 elas serão percorridas e, para cada regra, o processo de mediação irá calcular a média do uso dos últimos recursos ainda não processados. Na linha 7 acontece a coleta desses recursos, os quais estarão marcados com o atributo *processed*

como falso e, na própria consulta, é calculada a média desses dado (chamada *average(:value)*). Após isso é criado um *ResourceData* (recurso processado (mediado)), na linha 8 e, na linha 9, todos os recursos envolvidos nesse cálculo agora são marcados como processados.

4.5.4 Envio de dados de uso

Por fim, os dados mediados devem ser enviados ao serviço *aCCountS*, pois, de acordo com a arquitetura, é através desses dados que o uso da máquina será tarifado. O código 4.8 é o responsável por esse processo de envio.

Código 4.8: Protótipo do agente - Envio de dados processados ao *aCCountS*

```
1  def self.send_resources
2    resources = ResourceData.select([:value, 'resources_data.created_at', '
       rules.name']).where(uploaded: false).joins(:rule)
3    unless resources.empty?
4      size = Rule.where(active: true).size
5      resources2 = resources.map{|r| {"created_at"=>r.created_at,r.name=>r.
         value}}
6      res = []
7      resources2.each_slice(size) do |s|
8        elem = {}
9        s.each do |x|
10         elem.merge!(x)
11       end
12       res << elem
13     end
14     path = "/machines/#{ENV['MACHINE_ID']}/resources_data/set_ resources"
15     require 'net/http'
16     request = Net::HTTP::Post.new(path, initheader = {'Content-Type' =>'
         application/json'})
17     request.body = res.to_json
18     response = Net::HTTP.new(ENV['SERVER_IP'], ENV['SERVER_PORT']).start {|
         http| http.request(request) }
19     case response
20     when Net::HTTPSuccess
21       resources.update_all(uploaded: true)
22     end
23   end
24 end
```

Na linha 2, primeiramente todos os dados ainda não enviados são trazidos do banco para que possam ser preparados para envio (o campo *uploaded* serve para controlar isso). Se uma lista de recursos for retornado nessa consulta (teste da linha 3) então eles serão enviados, caso contrário, nada será feito.

Entre as linhas 4 a 13, todos os dados de recursos lidos com os seus respectivos nomes de regras são preparados para numa única requisição, todos serem enviados de uma vez. Isso é feito com a montagem de um pacote *JSON* listando os valores de recursos que foram mediados juntamente com o nome das regras correspondentes a cada um.

Na linha 14 é montado o caminho relativo para os quais esses dados serão enviados, que corresponde ao endereço com o identificador da máquina, na interface *i_resources*, dentro do controle de *resources_data* no serviço *aCCountS*. Após isso, a requisição é feita ao serviço com os dados sendo enviados, na linha 18. Por fim, caso haja uma resposta do serviço, identificando que a requisição foi processada com sucesso, o agente atualiza o atributo *uploaded* para verdadeiro em todos esses dados de recursos.

4.5.5 Temporização do agente

Todos os eventos descritos nas subseções anteriores poderiam acontecer a qualquer momento, no entanto, há que se configurar os instantes em que serão executados para não causar impacto nem no processamento da máquina virtual, nem na sua rede. Devido a isso, uma temporização padrão foi criada como segue, mas esses tempos podem ser alterado de acordo com a vontade do administrador:

- O processo de coleta de regras no serviço é feito uma vez por dia, normalmente de madrugada (no protótipo configurado para as 4:30). Sendo assim, mudanças nas regras só refletirão na medição das máquinas no dia posterior, o que é informado no serviço.
- O método de medição é feito de um em um minuto, sendo essa a granularidade mínima oferecida pelos agendadores de tarefas dos sistemas operacionais e suficientes para os propósitos do protótipo.
- O método de mediação é feito de hora em hora, de forma que os dados acumulados na mediação em uma hora (60 dados) serão usados para gerar um dado de mediação.
- A chamada ao processo de envio de dados é feita, a exemplo da coleta, uma vez ao dia, também no período da madrugada (também configurado às 4:30). Nesse momento o agente envia todas os dados de mediação computados naquele dia em uma só requisição (24 dados).

Vale lembrar ainda que todas essas tarefas são executadas como tarefas agendadas do sistema operacional de forma a executá-las como um serviço e não sobrecarregar o processamento da mesma.

4.5.6 Conclusões

O agente, que é instalado na máquina virtual para realizar os processamentos de medição e mediação, consome parte dos recursos da máquina virtual que está sendo monitorada. Entretanto, objetiva-se que o impacto deste monitoramento seja pequeno no desempenho da máquina virtual. Apesar dos agentes precisarem consumir recursos das máquinas virtuais, a periodicidade utilizada (1 minuto) na configuração dos agentes, não deve causar impacto negativo no processamento das atividades das máquinas virtuais, já que o *aCCountS-Agent* foi implementado por meio de tarefas agendadas no sistema operacional, com o objetivo de tornar

esse componente leve para a máquina virtual. Mesmo assim, testes de verificação no impacto que os agentes fornecem às máquinas virtuais podem ser tratados em trabalhos futuros.

Essa linguagem foi criada utilizando o *framework Xtext* para definir a gramática e gerar os analisadores léxico e sintático, mais detalhes sobre o assunto são encontrados no apêndice B

No caso particular de nossa proposta, o *DSLCompiler* teve uma grande influência do *Xtext*, um *framework open-source* para o desenvolvimento de DSLs. O *Xtext* permite que a partir da definição das regras de sintaxe da linguagem, sejam gerados seus analisadores léxico e sintático, além da personalização de um gerador de código, no qual pode-se associar os comandos da DSL desenvolvida com os de alguma outra linguagem de programação. Especificadamente para o *aCCounts*, escolheu-se Ruby, uma linguagem de crescente popularidade para desenvolvimento de aplicações Web de modo ágil e rápido. Porém, sua característica mais importante para nossa proposta é o fato da mesma interpretar dinamicamente seu código-fonte. Com isso, o código gerado pelo *Xtext* pode ser integrado naturalmente ao serviço, em tempo de execução, permitindo que modificações nas políticas sejam refletidas de imediato na tarifação dos recursos.

5 AVALIAÇÃO EXPERIMENTAL

Neste capítulo é detalhada a avaliação experimental do serviço proposto por meio do protótipo implementado. Esse protótipo serviu de base para aferir a corretude da arquitetura proposta e da DSL de tarifação, além de validar o uso do serviço com algumas plataformas de nuvens diferentes, nesse caso, Amazon AWS (AMAZON, 2013) e Windows Azure (AZURE, 2013). A aferição da corretude consiste em verificar primeiramente, se os dois componentes (agente e serviço) estão funcionando de forma correta, ou seja, os componentes estão realizando suas funcionalidades e trocando mensagens como apresentado na definição dos componentes e interfaces no capítulo 3. E, em segundo lugar, se as políticas, definidas pela linguagem, são compiladas corretamente, gerando uma função correspondente para o cálculo da cobrança. Isso será feito mostrando os experimentos efetuados com esse protótipo e os dados colhidos a partir deles.

5.1 Avaliação dos protótipos

Com o objetivo de avaliação do trabalho (*aCCountS*, *aCCountS-Agent* e *aCCountS-DSL*), testou-se a corretude do *(i)* fluxo de tarifação entre os dois componentes (agente e servidor) e do *(ii)* cálculo da fatura. Os ambientes de testes escolhidos foram a nuvem da *Amazon (Amazon EC2)* e a nuvem da *Microsoft*, *Windows Azure*.

Para a análise da proposta foram realizados experimentos com o *aCCountS*, com máquinas virtuais executando o *aCCountS-Agent* e com a *aCCountS-DSL*. Foram implantados máquinas virtuais nos centro de dados, cada uma associada a uma política de tarifação diferente. Por meio desses testes, buscou-se mostrar que o fluxo de tarifação entre os dois componentes (agente e servidor) ocorre de forma correta. Por meio das políticas também buscou-se verificar se a tarifação está sendo calculada de maneira correta, através do envio de dados de medição fictícios ao serviço, que permite comparar os resultados calculados pelo *aCCountS* com os valores esperados como resposta.

5.1.1 Avaliação do *aCCountS*

Para validação do fluxo de tarifação foram criadas três máquinas virtuais na *Amazon* e três máquinas virtuais no *Windows Azure*, onde foi implantado o *aCCountS-Agent*. Associadas a essas máquinas, foram criadas três combinações de política/perfil, de modo que se utilizou as mesmas configurações nas duas nuvens. As políticas definidas para cada nuvem são: *(i) simplesUsoSobPos*, *(ii) simplesTempoSobPos* e *(iii) simplesUsoRes1Pos* as quais serão detalhadas nas subseções a seguir.

5.1.1.1 Política *simplesUsoSobPos*

Essa política (Código 5.1) foi criada sendo definidos alguns requisitos embutidos em seu próprio nome, os quais são: forma de contabilização por ***uso*** *de CPU, memória e armazenamento*; garantia de alocação ***sob***-*demanda*; modelo de pagamento ***pós***-*pago*; perfil de máquina *pequena* (**simples**). Os preços definidos para o uso dos recursos são: uso CPU/h: $ 0.006, uso memoria/h: $ 0.006, uso armazenamento/h: $ 0.006 e taxa de uso da máquina ou centro de dados $ 0.14 ao mês.

Código 5.1: Definição da política *simplesUsoSobPos*

```
1  Policy simplesUsoSobPos{
2     var {
3        cpu; memoria; armazenamento; soma; taxaUso;
4     }
5     rules{
6        taxaUso = 0.14;
7        cpu = instance.cpu * $cpu;
8        memoria = instance.memoria * $memoria;
9        armazenamento = instance.armazenamento * $armazenamento;
10
11       soma = cpu + memoria + armazenamento + taxaUso;
12    }
13    return soma;
14
15 }
```

5.1.1.2 Política *simplesTempoSobPos*

A segunda política (Código 5.2) apresenta diferentes valores para os mesmos requisitos, tais como definidos pelo nome da política, como: forma de contabilização por ***Tempo***; garantia de alocação ***sob***-*demanda*; modelo de pagamento ***pós***-*pago*; perfil de máquina *pequena* (**simples**). O custo do uso da máquina definido para o experimento foi de: tempo de uso da máquina/h: $ 0.06.

Código 5.2: Definição da política *simplesUsoSobPos*

```
1  Policy simplesUsoRes1Pos{
2     var {
3        cpu; memoria; armazenamento; taxaUso; taxa; soma;
4     }
5     rules{
6        taxaUso = 0.14;
7        taxa = 0.0014;
8        cpu = instance.cpu * $cpu;
9        memoria = instance.memoria * $memoria;
10       armazenamento = instance.armazenamento * $armazenamento;
11
12       soma = cpu + memoria + armazenamento + taxa + taxaUso;
13    }
```

```
14      return soma;
15
16  }
```

5.1.1.3 Política *simplesUsoRes1Pos*

A terceira política (Código 5.3) utilizada no experimento, tal como as demais apresenta os requisitos: forma de contabilização por ***uso*** *de CPU, memória e armazenamento*; garantia de alocação reservada por 1 ano com taxa fixa de reserva no valor de $ 0.0014; modelo de pagamento ***pós****-pago* e perfil de máquina *pequena* (**simples**). O dos recursos definidos para essa política são: uso CPU/h: $ 0.0034, uso memória/h: $ 0.0034, uso armazenamento/h: $ 0.0034) e taxa de uso da máquina ou centro de dados $ 0.14.

Código 5.3: Definição da política *simplesUsoSobPos*

```
1  Policy simplesUsoRes1Pos{
2      var {
3          cpu; memoria; armazenamento; taxaUso; taxa; soma;
4      }
5      rules{
6          taxaUso = 0.14;
7          taxa = 0.0014;
8          cpu = instance.cpu * $cpu;
9          memoria = instance.memoria * $memoria;
10         armazenamento = instance.armazenamento * $armazenamento;
11
12         soma = cpu + memoria + armazenamento + taxa + taxaUso;
13     }
14     return soma;
15
16 }
```

5.1.1.4 Resultados do Experimento

As políticas definidas nesses experimentos possuem os preços dos recursos diferentes. Isso porque são levados em conta os requisitos definidos em cada política. No caso das políticas *simplesUsoSobPos* e *simplesUsoRes1Pos*, elas utilizam a mesma máquina pequena e o mesmo modelo pós-pago, entretanto o requisito garantia de alocação são diferentes, sendo os preços dos recursos para uma política sob-demanda maiores. Esses preços são baseados nos valores definidos para o tempo de uso de máquinas na *Amazon EC2* e o preço de uso de um recurso é um décimo do preço do tempo de máquina ligada. Essa relação foi definido fazendo-se um média da quantidade de recursos consumidos enquanto a máquina está ligada, exemplo de recursos: CPU, memória, armazenamento, upload, download, transação no banco de dados, transferência de dados da memória para o disco,entre outros.

As políticas *simplesUsoSobPos* e *simplesTempoSobPos* possuem todos os requisitos semelhantes com exceção do requisito *Forma de contabilização*, em que o primeiro é por Uso

e o segundo por Tempo, os preços definidos para o experimento, como definido, possui uma relação de 1 para 10. Esses valores não valem como comparação para identificar qual tipo de política apresenta melhores custos para o cliente ou provedora de nuvem, pois os valores dos preços foram definidos sem nenhuma base científica.

Os resultados encontrados a partir das configurações definidas estão apresentados na Tabela 5.1.

Tabela 5.1: Resultados dos experimentos realizados na *Amazon EC2* e *Windows Azure*

Política (Nuvem)	Registro de Medição	Reg. Cobrança
simplesUsoSobPos (Amazon)	{“cpu”: 0.01545, “memoria”: 0.80414, “armazenamento”: 2.7e-05}	$ 0.00492
simplesUsoRes1Pos (Amazon)	{“cpu”: 0.075, “memoria”: 0.8302205, “armazenamento”: 4.1e-05}	$ 0.00448
simplesTempoSobPos (Amazon)	{“tempoUso”: 1.0}	$ 0.06000
simplesUsoSobPos (Azure)	{“cpu”: 0.06428, “memoria”: 0.79215, “armazenamento”: 0.05129}	$ 0.00545
simplesUsoRes1Pos (Azure)	{“cpu”: 0.06851851851851852, “memoria”: 0.5443499818181821, “armazenamento”: 0.05128383636363637}	$ 0.00366
simplesTempoSobPos (Azure)	{“tempoUso":1.0}	$ 0.06000

No experimento, as seis máquinas virtuais foram executadas do dia 26 de junho de 2013, às 12h e 05 minutos até o dia 03 de julho, às 4h e 05 minutos e seus registros de medição (calculados a cada hora) utilizados para validação do fluxo de tarifação. Na tabela 5.1 são mostrados seis dos 960 registros de medição recebidos pelo *aCCountS-Service* (três de cada provedora de nuvem) e os registros de cobrança calculados. Dessa forma, verificou-se a corretude ao passo que os recursos medidos nas máquinas foram os previamente definidos no *aCCountS-Service* e os registros de faturamento foram calculados conforme as definições nas políticas e a partir dos dados enviados pelo agente.

5.1.2 Avaliação da *aCCountS-DSL*

Para validação da linguagem *aCCountS-DSL* foram criadas dez políticas de tarifação, combinando os diferentes requisitos apontados nesse trabalho. Cada política de tarifação foi testada com três conjuntos de valores fixos dos recursos associados a três casos de teste para o sistema. Desta forma, conhecia-se de antemão o valor esperado da fatura. Por meio de um *script*, esses dados foram enviados para o *aCCountS-Service* realizar a tarifação por meio das políticas definidas no *aCCountS-DSL*. Nas subseções abaixo estão definidas essas políticas.

5.1.2.1 Política *SobMedUsoPosPlus*

Na política *SobMedUsoPosPlus* (Código 5.4), são definidos os requisitos de tarifação. O requisito forma de contabilização de recursos, calcula separadamente o uso de memoria, CPU, armazenamento, transacaoBD e *upload*, definidos nas linhas 11, 12, 13, 14 e 15, respectivamente. Ela também fatura a quantidade de software e serviços utilizados, definidos na linha 30, no cálculo da conta. Também é cobrado uma taxa fixa pelo uso da central de dados, linha 9 e são dados descontos à fatura do usuário quando há economia de energia, dependendo da quantidade percentual de economia realizada. Das linhas 17 a 19 é verificado se a porcentagem de energia economizada pelo usuário é maior ou igual a 0.5%, se sim, o mesmo recebe um desconto na fatura de 0,03.

Também, descontos são dados aos clientes que consomem altas quantidades de memória e armazenamento. Das linhas 21 a 23 é verificado se a quantidade de memória utilizada pelo cliente é maior ou igual a 80%, se sim ele recebe um desconto de 0,04. Da linha 25 a 27, é definido um código que verifica a quantidade de armazenamento, caso ela seja maior ou igual a 80% esse desconto do usuário passa para 0.05. Nas linhas 29 e 30 os custos são somados e na linha 31 os descontos também, incluindo, nesse último, a porcentagem de multa por violação de SLA, por fim, na linha 32, o custo da fatura é calculado. O valor da taxa fixa pelo uso da central de dados foi definida de forma aleatória para o experimento e possui o valor de $ 0,14, o que corresponde a $ 100,00 por mês.

Código 5.4: Definição da política *SobMedUsoPosPlus*

```
1  Policy SobMedUsoPosPlus2 {
2     var {
3      taxaCentralDados; memoria; cpu; custo; armazenamento; transacaoBD;
4      upload; descontoEnergia; descontoUsuario; desconto;
5     }
6     rules{
7        descontoEnergia = 0;
8        descontoUsuario = 0;
9        taxaCentralDados = 0.14;
10
11       memoria = instance.memoria * $memoria;
12       cpu = instance.cpu * $cpu;
13       armazenamento = instance.armazenamento * $armazenamento;
14       transacaoBD = instance.transacaoBD * $transacaoBD;
15       upload = instance.upload * $upload;
16
17       if (instance.economiaEnergia >= 0.5 ){
18          descontoEnergia = 0.03;
19       }
20
21       if (instance.memoria >= 0.8 ){
22          descontoUsuario = 0.05;
23       }
24
25       if (instance.armazenamento >= 0.8 ){
26          descontoUsuario = 0.04;
```

```
27          }
28
29          custo = memoria + cpu + armazenamento + transacaoBD + upload
30       + instance.software + instance.servico + taxaCentralDados;
31          desconto = instance.sla + descontoEnergia + descontoUsuario;
32          custo = custo - custo * desconto;
33       }
34       return custo;
35    }
```

5.1.2.2 Política *Res1MedUsoPosPlus*

A política *Res1MedUsoPosPlus* estende a política *SobMedUsoPosPlus* reutilizando seu código, as variáveis definidas, a regras e seu retorno. A primeira política, que está sendo definida, possui uma regra de tarifação adicional, por sua garantia de alocação ser reservada, é pago uma taxa fixa de reserva mensal, então esse valor deve ser contabilizado para definir o custo do serviço. Com isso, na linha 6 é atribuído à variável o valor mensal de 0,0014 e o calculo do custo, na linha 7, é redefinido recebendo o valor do custo calculado pelas regras definidas na política *SobMedUsoPosPlus* adicionado ao valor da taxa fixa de reserva, retornando o novo valor do custo.

A diferença entre as políticas *Res1MedUsoPosPlus* e *SobMedUsoPosPlus* é o requisito garantia de alocação, esse não é definido por meio da linguagem *aCCountS-DSL* e sim por meio da configuração dos preços dos recursos, no caso, os valores para políticas reservadas são menores que para políticas sob-demanda.

Os preços definidos para o uso dos recursos são baseado no preços de tempo de utilização das máquinas virtuais definidas pela *Amazon EC2*. O preço de cada recurso corresponde a um décimo (1/10) do valor de tempo de utilização da máquina. Esse valor foi calculado tomando por média a quantidade de recursos que poderiam estar sendo consumidos enquanto uma máquina estivesse ligada, como por exemplo, CPU, memória, armazenamento, *upload*, *download*, transação no banco de dados, transações na memória, transações no HD, interrupções na CPU, realocação de espaço, entre outros.

Entretanto, esses valores não podem ser utilizados para comparar se uma política contabilizada por uso é mais econômica para o cliente que uma política contabilizada por tempo, ou vice-versa, pois esses valores são calculados sem base em pesquisas científicas.

No entanto, conclui-se que a utilização do comando *extend* torna a definição de políticas mais práticas e rápidas.

Código 5.5: Definição da política *Res1MedUsoPosPlus*

```
1  Policy Res1MedUsoPosPlus extends SobMedUsoPosPlus{
2     var {
3        custo; taxaFixa;
4        }
5     rules{
6        taxaFixa = 0.0014;
```

```
7         custo = custo + taxaFixa;
8     }
9     return custo;
10 }
```

5.1.2.3 Política *SobPeqUsoPosPlus*

A política *SobPeqUsoPosPlus* (Código 5.6) estende a política *SobMedUsoPosPlus* sem nenhuma diferenciação. O que distingue uma política da outra é o perfil da máquina, na primeira, é pequena e na segunda, é média. Esse requisito não é definido por meio da *aCCountS-DSL* e sim por meio da configuração dos preços dos recursos. Dessa maneira, a conta se torna diferente, mas a definição da política é a mesma. Entretanto ela foi definida para demonstrar o requisito perfil de máquina.

Código 5.6: Definição da política *SobPeqUsoPosPlus*

```
1  Policy SobPeqUsoPosPlus extends SobMedUsoPosPlus{
2     var {
3         custo;
4         }
5     return custo;
6  }
```

5.1.2.4 Política *SobMedTempoPosPlus*

A política *SobMedTempoPosPlus* (Código 5.7) não pode estender a política *SobMe-dUsoPosPlus*, pois a definição das regras de contabilização são bem diferentes. A contabiliza-ção da política *SobMedTempoPosPlus* é realizada pelo tempo de uso da máquina definido na linha 14. Por a cobrança calcular os valores medidos de hora em hora, o calculo do tempo de uso é 1, de 1 hora, multiplicado pelo preço correspondente a uma hora da máquina ligada, valor baseado na definição de preços na *Amazon EC2*.

Código 5.7: Definição da política *SobMedTempoPosPlus*

```
1  Policy SobMedTempoPosPlus{
2     var {
3         economiaEnergia; sla; descontoEnergia;
4         desconto; custo; software; servico;
5         tempoUso;
6     }
7     rules{
8         economiaEnergia = instance.economiaEnergia;
9         sla = instance.sla;
10        descontoEnergia = 0;
11
12        software = instance.software;
13        servico = instance.servico;
14        tempoUso = 1 * $tempoUso;
```

```
15
16          if (economiaEnergia >= 0.5 ){
17              descontoEnergia = 0.03;
18          }
19
20          custo = tempoUso + software + servico;
21          desconto = sla + descontoEnergia;
22          custo = custo - custo * desconto;
23      }
24      return custo;
25  }
```

5.1.2.5 Política *SobMedUsoPrePlus*

A política *SobMedUsoPrePlus* (Código 5.8) estende a política *SobMedUsoPosPlus*. A diferença entre as duas é o requisito modelo de tarifação, em que a primeira é pré-paga e a segunda é pós-paga. Entretanto esse requisito não é definido por meio da *aCCountS-DSL*, mas pela configuração dos preços dos recursos, o qual é maior para políticas pré-pagas. Com isso a definição da política é a mesma e sua definição visou testar o requisito modelo de tarifação.

Código 5.8: Definição da política *simplesUsoSobPos*

```
1  Policy SobMedUsoPrePlus extends SobMedUsoPosPlus{
2      var {
3          custo;
4      }
5
6      return custo;
7  }
```

5.1.2.6 Política *SobMedUsoPos*

A política *SobMedUsoPos* (Código 5.9) possui características semelhantes a política *SobMedUsoPosPlus*, entretanto não pode estendê-la, pois a segunda é mais complexa que a primeira, no entanto, a política *SobMedUsoPosPlus* poderia ser estendida da primeira. Isso não foi feito porque a política *SobMedUsoPos* foi definida posteriormente. Dessa forma é importante definir primeiramente as políticas mais simples para as mais complexas estenderem dessas e evitar o retrabalho.

Com isso, a política *SobMedUsoPos* realiza a contabilização por uso dos recursos, definidos da linha 18 a 22. Ela contabiliza, na linha 18, a memória, na linha 19, a CPU, na linha 20, o armazenamento, na linha 21, a transação do banco de dados e na linha 22, o *upload*.

A taxa de uso do *datacenter* é contabilizado e definido na linha 14, recebendo o valor mensal de $ 0.14. Nas linhas 15 e 16 são contabilizados os software e serviços consumidos pelo cliente. Nas linhas 24 e 25, por fim, é calculado o custo do serviço.

Código 5.9: Definição da política *simplesUsoSobPos*

```
1  Policy SobMedUsoPos{
2     var {
3        taxaUso; memoria; cpu; armazenamento;
4        transacaoBD; upload; custo;
5        software; servico;
6     }
7     rules{
8        taxaUso=0.14;
9        software = instance.software;
10       servico = instance.servico;
11
12       memoria = instance.memoria * $memoria;
13       cpu = instance.cpu * $cpu;
14       armazenamento = instance.armazenamento * $armazenamento;
15       transacaoBD = instance.transacaoBD * $transacaoBD;
16       upload = instance.upload * $upload;
17
18       custo = memoria + cpu + armazenamento + transacaoBD +
19             upload + software + servico + taxaUso;
20        }
21    return custo;
22 }
```

5.1.2.7 Política *Res1PeqUsoPosPlus*

A política *Res1PeqUsoPosPlus* (Código 5.10) estende a política *Res1MedUsoPosPlus*, sem acrescentar nenhuma regra de tarifação, pois a diferença entre as duas é o perfil de máquina, em que a primeira é pequena e a segunda é média.

Código 5.10: Definição da política *Res1PeqUsoPosPlus*

```
1  Policy Res1PeqUsoPosPlus extends Res1MedUsoPosPlus{
2     var {
3        custo;
4        }
5     return custo;
6  }
```

5.1.2.8 Política *Res1MedTempoPosPlus*

A política *Res1MedTempoPosPlus* (Código 5.11) estende a política *SobMedTempoPosPlus*. Por a primeira ser uma política reservada ela deve pagar uma taxa fixa de reserva, que a segunda política não paga, entretanto os preços do uso dos seus recursos são maiores. Na definição da política *Res1MedTempoPosPlus*, na linha 6 é definido o valor da taxa fixa de $ 0.0014 e esse valor é acrescentado ao custo contabilizado pelas regras estendidas da outra política. O valor da taxa fixa de reserva definido para os experimentos é baseado no valor cobrado pela mesma taxa na *Amazon EC2*, em que o cliente no experimento paga aproximadamente por

ano o valor de $ 367,92. Esse valor é calculado por $ 0.0014 que é pago por hora, multiplicado por 24 horas que resulta em $ 1,008, esse resultado é multiplicado por 365 dias o que retorna o valor pago no ano de $ 367,92.

Código 5.11: Definição da política *Res1MedTempoPosPlus*

```
1  Policy Res1MedTempoPosPlus extends SobMedTempoPosPlus{
2     var {
3        taxa; custo;
4     }
5     rules{
6        taxa = 0.0014;
7        custo = custo + taxa;
8     }
9     return custo;
10 }
```

5.1.2.9 Política *Res1MedUsoPrePlus*

A política *Res1MedUsoPrePlus* (Código 5.12) estende a política *Res1MedUsoPosPlus*. A definição das duas são iguais, sendo, portanto, a primeira definida pela extensão completa da outra. Isso ocorre porque a diferença entre as duas é apenas o requisito modelo de tarifação, definida por meio da configuração dos preços dos recursos.

Código 5.12: Definição da política *Res1MedUsoPrePlus*

```
1  Policy Res1MedUsoPrePlus extends Res1MedUsoPosPlus{
2     var {
3        custo;
4        }
5     return custo;
6  }
```

5.1.2.10 Política *Res1MedUsoPos*

A política *Res1MedUsoPos* (Código 5.13) estende da política *SobMedUsoPos*, a diferença entre as duas é a garantia de alocação, em que a primeira é reservada e precisa definir a taxa fixa de reserva, linha 6, que então é adicionado ao custo, calculado por meio das regras de cobrança definidas para a política *SobMedUsoPos*, a política estendida.

Código 5.13: Definição da política *Res1MedUsoPos*

```
1  Policy Res1MedUsoPos extends SobMedUsoPos{
2     var {
3        taxa; custo;
4     }
5     rules{
6        taxa = 0.0014;
7        custo = custo + taxa ;
```

```
8           }
9       return custo;
10  }
```

5.1.2.11 Realização do Experimento

No experimento foram utilizados três casos de testes (conjuntos de valores fixos dos recursos, os registros de medição) para verificar a corretude da tarifação em relação às diferentes políticas definidas. Cada política foi submetida aos três casos de teste, que são valores correspondentes aos registros de medição, dos quais as políticas foram submetidas e verificado se isso geraria uma conta com valor de acordo com o esperado. Estes casos de teste estão definidos no Tabela 5.2.

Tabela 5.2: Registros fixos usados como casos de teste

Casos de teste	Registro
CT1	{ “economiaEnergia”: 0.3, “memoria”: 0.5, “cpu”: 0.5, “armazenamento”: 0.5, “transacaoBD”: 0.5, “upload”: 0.5, “software”: 0.07, “servico”: 0.07, “sla”: 0.03}
CT2	{“economiaEnergia”: 0.5, “memoria”: 0.8, “cpu”: 0.8, “armazenamento”: 0.8, “transacaoBD”: 0.8, “upload”: 0.8, “software”:0.07, “servico”: 0.07, “sla”:0.0}
CT3	{“economiaEnergia”: 0.0, “memoria”: 0.3, “cpu”: 0.3, “armazenamento”: 0.3, “transacaoBD”: 0.3, “upload”: 0.3, “software”: 0.14, “servico”: 0.07, “sla”: 0.0}

Na Tabela 5.3 podem ser observados os resultados obtidos a partir da aplicação dos registros especificados como casos de testes mostrados na Tabela 5.2 em cada uma das políticas descritas anteriormente.

Tabela 5.3: Resultados obtidos no experimento

	Caso de Teste 1		Caso de Teste 2		Caso de Teste 3	
Política	Previsto	Obtido	Esperado	Obtido	Esperado	Obtido
SobMedUsoPosPlus	0.30070	0.30070	0.30504	0.30504	0.36800	0.36800
Res1MedUsoPosPlus	0.28949	0.28949	0.28710	0.28710	0.36160	0.36160
SobPeqUsoPosPlus	0.28615	0.28615	0.28272	0.28272	0.35900	0.35900
SobMedTmpPosPlus	0.25220	0.25220	0.25220	0.25220	0.33000	0.33000
SobMedUsoPrePlus	0.32980	0.32980	0.34968	0.34968	0.38600	0.38600
SobMedUsoPos	0.31000	0.31000	0.32800	0.32800	0.36800	0.36800
Res1PeqUsoPosPlus	0.28125	0.28125	0.27445	0.27445	0.35650	0.35650
Res1MedTmpPosPlus	0.20316	0.20316	0.20316	0.20316	0.27940	0.27940
Res1MedUsoPrePlus	0.30598	0.30598	0.31239	0.31239	0.37180	0.37180
Res1MedUsoPos	0.29840	0.29840	0.30860	0.30860	0.36160	0.36160

Pelo resultado do experimento, constatou-se que em todos os casos de teste, o serviço calcula o valor de tarifação como esperado. Desta forma, conclui-se que o cálculo da fatura

é realizado de forma correta, e que a linguagem *aCCountS-DSL* pode ser utilizada com segurança para tarifação de serviços de infraestrutura em computação em nuvem, definindo regras flexíveis.

5.1.3 Conclusões

O agente, que é instalado na máquina virtual para realizar os processamentos de medição e mediação, consome parte dos recursos da máquina virtual que está sendo monitorada. Entretanto, objetiva-se que o impacto deste monitoramento seja pequeno no desempenho da máquina virtual. Apesar dos agentes precisarem consumir recursos das máquinas virtuais, a periodicidade utilizada (1 minuto) na configuração dos agentes, não deve causar impacto negativo no processamento das atividades das máquinas virtuais, já que o *aCCountS-Agent* foi implementado por meio de tarefas agendadas no sistema operacional, com o objetivo de tornar esse componente leve para a máquina virtual. Mesmo assim, testes de verificação no impacto que os agentes fornecem às máquinas virtuais podem ser tratados em trabalhos futuros.

A linguagem *aCCountS-DSL* foi criada utilizando o *framework Xtext* para definir a gramática e gerar os analisadores léxico e sintático, mais detalhes sobre o assunto são encontrados no apêndice B

No caso particular de nossa proposta, o *DSLCompiler* teve uma grande influência do *Xtext*, um *framework open-source* para o desenvolvimento de DSLs. O *Xtext* permite que a partir da definição das regras de sintaxe da linguagem, sejam gerados seus analisadores léxico e sintático, além da personalização de um gerador de código, no qual pode-se associar os comandos da DSL desenvolvida com os de alguma outra linguagem de programação. Especificadamente para o *aCCounts*, escolheu-se Ruby, uma linguagem de crescente popularidade para o desenvolvimento de aplicações Web de modo ágil. Porém, sua característica mais importante para nossa proposta é o fato da mesma interpretar dinamicamente seu código-fonte. Com isso, o código gerado pelo *Xtext* pode ser integrado naturalmente ao serviço, em tempo de execução, permitindo que modificações nas políticas sejam refletidas de imediato na tarifação dos recursos.

6 CONCLUSÕES E TRABALHOS FUTUROS

Computação em nuvem é um termo bastante recente, muitas pesquisas estão sendo realizadas para amadurecer essa tecnologia. A tarifação em nuvem é uma das principais características desse paradigma e muitos estudos tem sido realizados nesse campo, sendo que estudá-la requer a pesquisa de novas formas de cobrança pelos serviços da nuvem. Esse trabalho procurou melhorar as soluções existentes através da proposta de um serviço de tarifação flexível e desacoplado da infraestrutura de nuvem como definido por meio de sua arquitetura (diagrama de componentes) e do protótipo criado e avaliado.

Para definir esse serviço foram realizadas pesquisas de artigos acadêmicos, especialmente, na área de tarifação em nuvem e foi estudado a documentação do modelo de tarifação das principais representantes da indústria de computação em nuvem. Com isso verificou-se a necessidade de unir as propostas da academia com os modelos da indústria em uma só solução.

Definiu-se, portanto, um modelo de cobrança, que abrangia diferentes requisitos vindos tanto da academia quanto das indústrias. Atender a esses requisitos foi a base para a criação da proposta. Em segundo lugar, percebeu-se que as plataformas para desenvolvimento de nuvens privadas não ofereciam suporte a tarifação em nuvem e muitas provedoras desenvolviam suas políticas diretamente em suas infraestruturas, tornando difícil a manutenção e criação de novas políticas de cobrança.

Criou-se, para isso, uma DSL que permite definir facilmente políticas de tarifação que atendem aos diversos requisitos catalogados por meio dos estudos e também criou-se um serviço de tarifação, que utiliza essa DSL para criar e atualizar essas políticas de forma fácil, prática e rápida.

Para definir o serviço de tarifação, foi utilizando como base o fluxo de contabilização proposto por Ruiz-Agundez (**??**). Esse fluxo é definido por funções que trabalham de maneira sequencial para monitorar e calcular a conta a ser paga pelo cliente da nuvem.

Por meio da pesquisa, concluiu-se que cada provedora de nuvem possui sua implementação de infraestrutura própria, exigindo, portanto, um componente de tarifação também flexível para atender diferentes provedoras com diferentes sistemas operacionais. Com isso o fluxo de tarifação pesquisado foi modificado para atender as novas exigências do sistema, cujo fluxo foi utilizado para criação da arquitetura do serviço proposto.

Com a arquitetura definida e os requisitos de tarifação definidos foi criada a linguagem de tarifação e acoplada ao serviço de tarifação proposto, para que o mesmo utilizasse a DSL a fim de atender às diversas necessidades da tarifação em nuvem e propor um serviço novo e com uma certa flexibilidade para que atendesse diferentes regras de negócios das provedoras de nuvem e que elas pudessem utilizá-las, independente da infraestrutura de suas nuvens.

Com isso, o serviço de tarifação foi implementado junto com um agente de tarifação para monitorar a utilização dos serviços na nuvem. Esse agente foi criado de forma a ser leve o bastante para não sobrecarregar a nuvem utilizada pelo cliente com seu processamento. Dessa forma, o agente se comunica com o serviço a fim de realizar o fluxo completo da tarifação.

Por meio do serviço e do agente implementados foram realizados testes para validar a corretude do serviço e da linguagem. Para isso, o serviço foi ativado e o agente foi implantado em diferentes nuvens com a finalidade de verificar se o serviço de tarifação realiza sua funcionalidade de forma correta. Por meio dos testes realizados concluiu-se que o serviço realiza o processo de forma correta. Para validação da linguagem, foram criadas diferentes políticas e o serviço recebeu como entrada registros de medição previamente definidos, assim como o resultado do cálculo final da tarifação. Para verificar a corretude na linguagem foi verificado se os valores da cobrança contabilizados pelo serviço por meio da política definida resultavam nos valores previamente calculados. Concluiu-se, portanto, que o serviço realiza a tarifação de acordo com as regras definidas na linguagem.

Com isso, o objetivo do trabalho de criar um modelo de tarifação em nuvens flexível foi conseguido e como contribuição foram publicados dois artigos, um no Workshop de Computação em Clouds e Aplicações (WCGA), SBRC-2013 e outro no Simpósio Brasileiro de Componentes, Arquiteturas e Reutilização de Software (SBCARS), CBSOFT-2013.

Como trabalhos futuros vislumbra-se a extensão do serviço para atender requisitos de segurança, disponibilidade e escalabilidade, que são de suma importância para utilização de serviço de tarifação em nuvem. Além disso, sugere-se como trabalho futuro a automatização na criação dos agentes nas plataformas de nuvem, a utilização de serviços de monitoramento mais robustos como o *Ganglia*, a criação de componentes para monitorar as configurações nas máquinas virtuais e quantificar a economia de energia realizada, assim como quantificar a SLA e gerenciar o modelo de tarifação (pré-pago e pós-pago).

REFERÊNCIAS BIBLIOGRÁFICAS

AMAZON. *Amazon Elastic Compute Cloud (Amazon EC2)*. 2013. Website. Http://aws.amazon.com/pt/ec2/.

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

AZURE, W. *A nuvem para a empresa moderna*. 2013. Website. Http://www.windowsazure.com/pt-br/.

BUYYA, R.; BROBERG, J.; GOSCINSKI, A. *Cloud computing : principles and paradigms*. [S.l.]: John Wiley Sons, Inc., 2011.

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>.

CARACAS, A.; ALTMANN, J. A pricing information service for grid computing. In: *Proceedings of the 5th international workshop on Middleware for grid computing: held at the ACM/IFIP/USENIX 8th International Middleware Conference*. [S.l.]: ACM, 2007.

CHIRIGATI, F. S. *Computação nas Nuvens*. 2012. Website. Http://www.gta.ufrj.br/ensino/eel879/trabalhos_v1_2009_2/seabra/index.html.

DANTAS, M. *A lógica do capital-informação: a fragmentação dos monopólios e a monopolização dos fragmentos num mundo de comunicações globais*. [S.l.: s.n.], 2002.

ELMROTH, E. et al. Accounting and billing for federated cloud infrastructures. In: *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*. [S.l.: s.n.], 2009.

GOOGLE. *Google App Engine*. 2013. Website. Https://developers.google.com/appengine/?hl=pt-BR.

GROUP, H. S. *Business Rule Management System*. 2012. Website. Http://www.hartmannsoftware.com/pub/Enterprise-Rule-Applications/brms.

IBM. *IBM Smart Cloud*. 2012. Website. Http://www-03.ibm.com/marketing/br/smarterplanet/cloud/.

IBM. *Cloud billing service, An SOA-enabled billing service module for the cloud environment*. 2013. Website. Http://www.ibm.com/developerworks/cloud/library/cl-devcloudmodule.

IMADA, T.; SATO, M.; KIMURA, H. Power and qos performance characteristics of virtualized servers. In: *Grid Computing, 2009 10th IEEE/ACM International Conference on*. [S.l.: s.n.], 2009.

I.R., A.; Y.K., P.; P.G., B. Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requirements Engineering*, Springer London, 2006.

JBILLING. *jBilling*. 2013. Website. Http://www.jbilling.com/.

JSON. *Introducing JSON*. 2013. Website. Http://www.json.org/.

LICKLIDE, J. *Memorandum For Members and Affiliates of the Intergalactic Computer Network*. 2012. Website. Http://goo.gl/2xpb0.

LIVESTREAM. *Hybrid Clouds - The Best of Both Worlds*. 2013. Website. Www.livestream.com/gigaomtv/video?clipId=pla_7a8babfe-7b50-472e-bbb8-1619d153ada4&utm_source=lslibrary&utm_medium=ui-thumb.

LIVESTREAM. *Structure2010-Day2 - Hybrid Clouds - The Best of Both Worlds?* 2013. Website. Www.livestream.com/gigaomtv/video?clipId=pla_7a8babfe-7b50-472e-bbb8-1619d153ada4&utm_source=lslibrary&utm_medium=ui-thumb.

MELL, P.; GRANCE, T. The nist definition of cloud computing. *National Institute of Standards and Technology*, NIST, 2009.

MELL, P. M.; GRANCE, T. *SP 800-145. The NIST Definition of Cloud Computing*. Gaithersburg, MD, United States, 2011.

MONGODB. *mongoDB*. 2013. Website. Http://www.mongodb.org/.

NARAYAN, A. et al. Smart metering of cloud services. In: *Systems Conference (SysCon), 2012 IEEE International*. [S.l.: s.n.], 2012. p. 1 –7.

NETFLIX. *Filmes e Séries de TV*. 2013. Website. Https://signup.netflix.com/home?country=1&rdirfdc=true.

RDIO. *rdio*. 2013. Website. Http://www.rdio.com/.

RUBY. *Ruby - A Programmer's Best Friend*. 2013. Website. Https://www.ruby-lang.org/pt/.

RUIZ-AGUNDEZ, I.; PENYA, Y.; BRINGAS, P. A flexible accounting model for cloud computing. In: *SRII Global Conference (SRII), 2011 Annual*. [S.l.: s.n.], 2011.

SILVA, E. da; LUCREDIO, D. Software engineering for the cloud: A research roadmap. In: *Software Engineering (SBES), 2012 26th Brazilian Symposium on*. [S.l.: s.n.], Sept.

SILVA, F. A. P. da et al. Accounting models in cloud and grid computing: A systematic mapping study. In: *Proceedings of the 2012 International Conference on Grid Computing and Applications*. [S.l.: s.n.], 2012. (GCA '12).

SOTOMAYOR, B. et al. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 13, n. 5, p. 14–22, sep 2009. ISSN 1089-7801. Disponível em: <http://dx.doi.org/10.1109/MIC.2009.119>.

SQLITE. *SQLite*. 2013. Website. Http://www.sqlite.org/.

STEAM. *Jogos*. 2013. Website. Http://store.steampowered.com/.

STORE iTunes. *iTunes Store*. 2013. Website. Http://store.apple.com/br/.

VAQUERO, L. M. et al. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 39, dez. 2008. ISSN 0146-4833.

XTEND. *Java 10, Today!* 2013. Website. Http://www.eclipse.org/xtend/.

XTEXT. *Language Development Made Easy*. 2013. Website. Http://www.eclipse.org/Xtext/.

YU, Y.; BHATTI, S. Energy measurement for the cloud. In: *Parallel and Distributed Processing with Applications (ISPA), 2010 International Symposium on*. [S.l.: s.n.], 2010.

APÊNDICE A – GRAMÁTICA DA *ACCOUNTS-DSL*

No código A.1 tem-se a gramática da linguagem *aCCountS-DSL* desenvolvida com as definições suportadas por essa linguagem:

Código A.1: Gramática da aCCountS-DSL

```
1  grammar br.ufc.great.politica.Politica with org.eclipse.xtext.common.
       Terminals
2  generate politica "http://www.xtext.org/politica/Politica"
3
4  Model: policy = Policy;
5  Policy:
6      'Policy' name = ID ('extends' supertype = ID)? '{'
7    ('var' '{' variable+=Variable* '}' )?
8          ('rules' '{' rules+=Rules* '}' )?
9          'return' returning = Return
10      '}'
11 ;
12 Variable: name=ID ':' type=Number ';';
13 Number: 'float' | 'double';
14 Rules: ifstmt=IfStatement | exp=Expression;
15 Expression: name=[Variable] '=' operation=Operation ';';
16 Operation: term=Term re += SubOperation* ;
17 SubOperation: op='+' term=Term | op='-' term=Term ;
18 Term: factor=Factor rt += SubTerm* ;
19 SubTerm: op='*' factor = Factor | op='/' factor = Factor;
20 Factor: name=[Variable] | price=Price | instance=Instance | number=Num | "(
       " operation=Operation ")";
21 Price: '$'name=ID;
22 Instance: 'instance.'name=ID;
23 Num: INT | INT '.' INT;
24 Return: operation=Operation ';';
25 IfStatement:
26     'if' '(' condition=Condition ')' '{' thenrules += (Rules)*
27      ('}' 'else' '{' elserules += (Rules)*)? '}'
28 ;
29 Condition returns Condition: notexp=NotExpr | logexp=LogicExpr;
30 NotExpr: 'not' value=Condition;
31 LogicExpr: orexp=OrExpression;
32 OrExpression: left1=AndExpression (=>'or' {OrExpression.left=current} right
       =Condition)?;
33 AndExpression: left1=BooleanCondition (=>'and' {AndExpression.left=current}
        right=Condition)?;
34 BooleanCondition: left1=Operation2 ({BooleanCondition.left=current} comp=
       Comparation right=Operation2)?;
35 Comparation: '<' | '>' | '==' | '!=' | '>=' | '<=';
36 Operation2: term=Term2 re += SubOperation2* ;
37 SubOperation2: op='+' term=Term2 | op='-' term=Term2;
38 Term2: factor=Factor2 rt += SubTerm2*;
39 SubTerm2: op='*' factor = Factor2 | op='/' factor = Factor2;
```

```
40 Factor2: name=[Variable] | price=Price | instance=Instance | number=Num |
       parexp=ParenthesizedExpr;
41 ParenthesizedExpr: '(' value=Condition ')';
```

Nessa gramática, é possível notar os principais requisitos da DSL sendo atendidos, na medida em que propicia a criação de regras contendo apenas cálculos matemáticos e o retorno de um valor que será o calculado na política, além da possibilidade de fazer referência a variáveis no formato **'instance.recurso'**, que farão referência a um recurso usado na máquina virtual e **'$recurso'** que referenciarão preços dos recursos definidos nos perfis de máquinas virtuais. A seguir são mostrados mais detalhes sobre os comandos escolhidos para a DSL e a sintaxe do Xtext para a definição dos mesmos.

- Na linha 1 tem-se a declaração do nome da linguagem (*br.ufc.great.politica.Politica*), seguindo a sintaxe do Xtext e especificando de qual gramática base ela irá herdar suas definições (*org.eclipse.xtext.common.Terminals*). Esse nome segue o padrão Java de especificação de pacotes e determina em que pasta deverão estar os códigos-fonte.

- A linha 2 pode ser entendida como: gere um EPackage (grafos de objetos em memória criados pelos analisadores do Xtext enquanto consome o texto) com o nome *politica* e a URI base "http://www.xtext.org/politica/Politica". Essa definição é obrigatória na definição de gramáticas usando Xtext.

- Na linha 4 começa-se a definição da gramática propriamente dita, com a especificação do modelo base: *Policy*. Nessa e nas regras subsequentes, há uma "variável" (*policy*=) apontando para os não-terminais de forma que esse elemento possa ser capturado e processado durante a geração de código. A finalização de cada definição em Xtext é feita por um ponto-e-vírgula (;).

- Da linha 5 a linha 11 tem-se a estrutura geral de uma política. Essas regras especificam que a definição da política começa com a palavra *Policy* seguida do nome da politica, identificada por *ID* que, no Xtext, representa um identificador (tokens de texto que nomeiam entidades da linguagem, sendo também chamados "símbolos"). Esse nome pode ou não ser seguido da palavra *extends*, o que indicará que a política herda definições de outra política, com o nome dessa política aparecendo logo a frente e, também sendo definido como um identificador. Essa primeira linha finaliza com um "{" que marca a abertura do código da política.

  As linhas 7, 8 e 9 determinam as seções de código que a política terá: seção de variáveis, seção de regras e seção de retorno. As duas primeiras são opcionais, indicadas pelo símbolo "?" ao final da linha, já a última é obrigatória, pois a política sempre deverá retornar algum valor. Dessa forma, a política pode conter apenas a seção de retorno, suprimindo as seções de variáveis e regras.

  A seção de variáveis começa com a palavra-chave "var", seguida de "{" que delimita esta seção. Logo após virão as definições de variáveis usando a regra "Variable" uma ou mais vezes (indicado pelo "*" logo após o não-terminal). Da mesma forma que o nome da

politica, atribui-se a “variable” esse comando para que a lista de definições de variáveis seja capturada na geração de código. Por fim, a seção de variáveis se encerra com “}”.

De forma análoga, a seção de regras começa com a palavra-chave “rules” com um “{” subsequente delimitando o início da seção. As regras são definidas pelo não-terminal “Rules” uma ou mais vezes e capturados pela variável “rules” em tempo de geração de código. Por fim, também se encerra essa seção com um “}”.

A última seção, seção de retorno, é definida em uma linha somente com a palavra-chave “return” seguida do valor de retorno. Esse valor de retorno é especificado através do não-terminal “Return”, que é capturado na geração de código pela variável “returning”.

A linha 9 determina o fechamento da definição da política, através de um “}” correspondente à chave de abertura.

- A linha 12 possui a regra de definição de uma variável, que deve ser começada pelo nome da variável (um identificador), seguido de “:” e do tipo da mesma, especificado através do não-terminal *Number*. Finaliza-se a declaração de uma variável com “;”.

- Na linha 13 tem-se a regra *Number* que especifica que o tipo das variáveis pode ser “float” ou “double”, visto que representarão, em geral, valores monetários, necessita-se que as variáveis sejam ponto flutuante.

- *Rules*, na linha 14, determina que as regras poderão ser de dois tipos: *IfStatement* para o caso de um comando de seleção, ou *Expression*, para o caso de regras simples de cálculos matemáticos.

- Na linha 15, a regra *Expression* determina que uma expressão deverá começar com um nome correspondente a alguma das variáveis definidas na seção de variáveis. Logo em seguida, o sinal de igualdade (“=”) deve aparecer para indicar qual valor será atribuído à mesma. Por fim, esse valor é determinado pelo não-terminal *Operation* e seguido de “;”.

- A regra *Operation*, definida na linha 16, especifica que uma operação deve começar com um termo (*Term)* e seguida de zero (“*”) ou mais sub-operações (*SubOperation*).

- A linha 17 mostra a regra *SubOperation* com duas opções, o sinal de adição (“+”) ou subtração (“-”), ambos seguidos por um termo (*Term*).

- Na linha 18, um termo *Term* é definido como uma sequência contendo primeiro um fator (*Factor*) e logo em seguida um conjunto de zero (“*”) ou mais sub-termos (*SubTerm*).

- Similar à *SubOperation*, um sub-termo (*Subterm*) é definido com duas opções (linha 19), sinal de multiplicação (“*”) ou sinal de divisão (“/”) seguido de um fator (*Factor*).

- Na linha 20, a regra *Factor* é definida com 5 opções: (i) uma variável, obrigatoriamente dentre as definidas na seção de variáveis; (ii) um não-terminal *Price*, remetendo a preços de recursos; (iii) um não-terminal *Instance*, referenciando dados de uso nas máquinas; (iv) um não-terminal *Num*, definindo um número (inteiro ou ponto flutuante); ou (v) uma nova operação (*Operation*) entre parêntesis, para suportar parentização de expressões aritméticas.

•Na linha 21, a definição da regra *Price* mostra que a linguagem aceita variáveis de precificação no formato "$ + identificador", que significa que terão que começar com "$" e seguir com um identificador.

•De forma similar, na linha 22, a definição de *Instance* representa os recursos medidos nas máquinas virtuais que terão o formato "instance. + identificador" ou seja, sempre começando com o pré-fixo "instance." seguido de um identificador. Essa regra e a anterior fazem com que a linguagem suporte qualquer nome de recurso, com preço e medição na máquina podendo ser escritos segundo essas definições.

•Na linha 23, tem-se a definição dos formatos de números aceitos, que serão inteiro e ponto flutuante. A definição *INT* em Xtext representa o conjunto de caracteres numéricos (0 a 9) em qualquer quantidade, assim, nessa definição podemos representar os números inteiros ou números com casas decimais.

•O comando *Return*, na linha 24, representa o retorno da política. Sua definição conta com a mesma estrutura de uma operação (Operation), assim, pode-se ter cálculos matemáticos como retorno, assim como apenas valores de variáveis. Ao final do comando deve haver um ponto e vírgula (";").

•Entre as linhas 25 a 28, está definido o comando de seleção (*IfStatement*), que dá à linguagem mais dinamicidade na aplicação das regras das políticas. Essa estrutura é criada com a palavra-chave "*if*" com a condição de seleção (*Condition*) entre parêntesis. Após à condição estão os comandos que serão executados caso a condição seja verdadeira delimitados por chaves. Esses comandos são zero ou mais não terminais do tipo *Rules* ("*(Rules)**"). Logo em seguida, pode haver ou não cláusula *else*, definindo comandos da mesma forma que anteriormente para serem executados caso a condição avaliada seja falsa, também entre chaves.

•A linha 29 representa o formato de condição aceito pela linguagem, que pode ser uma expressão de negação *NotExpr* ou uma expressão lógica *LogicExpr*. O comando *returns Condition* é uma forma de informar ao Xtext o que deve ser retornado nessa expressão, para que, nessa avaliação, não haja ambiguidades.

•A regra *NotExpr*, na linha 30, define de forma simples uma negação com a introdução da palavra *not*, seguida de uma condição *Condition*.

•Na linha 31, a regra *LogicExpr* começa a definição da expressão lógica pelo conectivo *or* (*OrExpression)*, para que tenha menor prioridade.

•A regra OrExpression, na linha 32, define que o operação de *or* lógico deve conter outra *OrExpression* do lado esquerdo, a palavra-chave "*or*" e uma condição (*Condition*) do lado direito. Se não houver conectivo na expressão, será passado diretamente para a regra *AndExpression*.

•De forma idêntica, uma *AndExpression* (linha 33) é definida por outra *AndExpression* do lado esquerdo, o conectivo "*and*" e uma condição (*Condition)* do lado direito. Também,

se não houver o conectivo na expressão, será passado direto para a regra *BooleanCondition*. Essa regra e a anterior, definidas nesse formato, permitem qualquer combinação de expressões lógicas, conservando prioridade menos para o operador "*or*".

- O item *BooleanCondition* (linha 34) permite a definição de expressões relacionais, de forma bem parecida com os operadores anteriores ("*or*" e "*and*"). No lado esquerdo na relação estará outra *BooleanCondition*, no meio o operador de comparação (*Comparation*) e na direita outras operações definidas pela regra *Operation2*.

- A regra *Comparation*, na linha 35, define os operadores de comparação possíveis de serem usados na linguagem.

- As regras nas linhas 36 a 40 são cópias das regras *Operation*, *SubOperation*, *Term*, *SubTerm* e *Factor*, que precisaram ser feitas para permitir o uso de expressões aritméticas nas expressões lógicas sem introduzir ambiguidades com as regras anteriormente definidas.

- Por fim, a regra *ParenthesizedExpr*, na linha 41, permite a parentização de expressões lógicas para modificação de prioridades na especificação de condições no comando de seleção.

APÊNDICE B – GERADOR DE CÓDIGO DO *DSLCOMPILER*

O Código B.1 mostra a implementação completa do gerador de código do *DSLCompiler*, responsável por receber os *tokens* processados a partir de um código escrito na aCCountS-DSL e gerar código *Ruby*.

Código B.1: Gerador de código implementado com Xtend

```
1  class PoliticaGenerator implements IGenerator2 {
2
3    @Inject extension IQualifiedNameProvider
4    override void doGenerate(Resource resource, IFileSystemAccess fsa) {
5      for(e: resource.allContents.toIterable.filter(typeof(Policy))) {
6            fsa.generateFile(
7              e.name.toFirstUpper + ".java", e.compile)
8        }
9    }
10
11   override CharSequence doGenerateS(Resource resource) {
12     return resource.allContents.toIterable.filter(typeof(Policy)).head.
         compile;
13   }
14
15   def compile(Policy p) '''
16     class <<p.name.toFirstUpper>><<IF p.supertype!= null>> < <<p.supertype
         >> <<ENDIF>>
17       def self.execute(machine, instances = {})
18         <<IF p.supertype!= null>>super<<ENDIF>>
19         parameters = {}
20         prices = []
21         <<FOR ps:p.eAllContents.toIterable.filter(typeof(Price))>>
22           prices << "<<ps.name>>"
23         <<ENDFOR>>
24         parameters = JSON.parse(machine.status(prices))
25         parameters["instances"] = instances
26         <<FOR f:p.rules>>
27           <<f.compile>>
28         <<ENDFOR>>
29         <<p.returning.compile>>
30       end
31
32       <<FOR vars:p.variable>>
33         attr_accessor :<<vars.name>>
34       <<ENDFOR>>
35     end
36   '''
37
38   def compile(Rules r) {
39     '''<<IF r.ifstmt != null>><<r.ifstmt.compile>><<ELSE>><<r.exp.compile
         >><<ENDIF>>'''
40   }
```

```
41
42    def compile(Expression e){
43      '''@<<e.name.name>> = <<e.operation.compile>>'''
44    }
45
46    def compile(Operation op){
47      '''<<op.term.compile>><<FOR sub:op.re>><<sub.compile>><<ENDFOR>>'''
48    }
49
50    def compile(SubOperation ops){
51      ''' <<ops.op>> <<ops.term.compile>>'''
52    }
53
54    def compile(Term t) {
55      '''<<t.factor.compile>><<FOR sub:t.rt>><<sub.compile>><<ENDFOR>>'''
56    }
57
58    def compile(SubTerm ts){
59      ''' <<ts.op>> <<ts.factor.compile>>'''
60    }
61
62    def compile(Factor f) {
63      '''<<IF f.name != null>> @<<f.name.name>><<ELSEIF f.price != null>>
            parameters["prices"]["<<f.price.name>>"]<<ELSEIF f.instance != null
            >> parameters["instances"]["<<f.instance.name>>"]<<ELSEIF f.
            operation != null>>(<<f.operation.compile>>)<<ELSE>> <<f.number>><<
            ENDIF>>'''
64    }
65
66    def compile(Return r){
67      '''<<r.operation.compile>>'''
68    }
69
70    def compile(IfStatement i) {
71      '''
72      if (<<i.condition.compile>>)
73        <<FOR r:i.thenrules>><<r.compile>><<ENDFOR>>
74      <<IF i.elserules.size > 0>>
75      else
76        <<FOR r:i.elserules>><<r.compile>><<ENDFOR>>
77      <<ENDIF>>
78      end
79      '''
80    }
81
82    def compile(Condition c) {
83      '''<<IF c.notexp != null>><<c.notexp.compile>><<ELSE>><<c.logexp.
            compile>><<ENDIF>>'''
84    }
85
86    def compile(ParenthesizedExpr expr) {
87      '''(<<expr.value.compile>>)'''
```

```
88    }
89
90    def compile(NotExpr expr) {
91      '''not <<expr.value.compile>>'''
92    }
93
94    def compile(LogicExpr expr) {
95      '''<<expr.orexp.compile>>'''
96    }
97
98    def compile(OrExpression o) {
99      '''<<IF o.left1 != null>><<o.left1.compile>><<ENDIF>><<IF o.right !=
            null>><<o.left.compile>> or <<o.right.compile>><<ENDIF>>'''
100   }
101
102   def compile(AndExpression a) {
103     '''<<IF a.left1 != null>><<a.left1.compile>><<ENDIF>><<IF a.right !=
            null>><<a.left.compile>> and <<a.right.compile>><<ENDIF>>'''
104   }
105
106   def compile(BooleanCondition bc) {
107     '''<<IF bc.left1 != null>><<bc.left1.compile>><<ENDIF>><<IF bc.right !=
             null>><<bc.left.compile>> <<bc.comp>> <<bc.right.compile>><<ENDIF
            >>'''
108   }
109
110   def compile(Operation2 op){
111     '''<<op.term.compile>><<FOR sub:op.re>><<sub.compile>><<ENDFOR>>'''
112   }
113
114   def compile(SubOperation2 ops){
115     ''' <<ops.op>> <<ops.term.compile>>'''
116   }
117
118   def compile(Term2 t) {
119     '''<<t.factor.compile>><<FOR sub:t.rt>><<sub.compile>><<ENDFOR>>'''
120   }
121
122   def compile(SubTerm2 ts){
123     ''' <<ts.op>> <<ts.factor.compile>>'''
124   }
125
126   def compile(Factor2 f) {
127     '''<<IF f.parexp != null>><<f.parexp.compile>><<ELSEIF f.name != null>>
             @<<f.name.name>><<ELSEIF f.price != null>> parameters["prices"]["<<
            f.price.name>>"]<<ELSEIF f.instance != null>> parameters["instances"
            ]["<<f.instance.name>>"]<<ELSE>> <<f.number>><<ENDIF>>'''
128   }
129
130 }
```

A interface *IGenerator2*, a qual a classe do gerador implementa, não é padrão do Xtend. Ela foi criada para ser possível a execução do compilador com retorno apenas em texto, e não arquivo. Essa interface herda da *IGenerator*, que é a original do Xtend, e pode ser vista no Código B.2

Código B.2: Interface *IGenerator2*

```
1  package br.ufc.great.politica;
2
3  import org.eclipse.emf.ecore.resource.Resource;
4  import org.eclipse.xtext.generator.IGenerator;
5
6  public interface IGenerator2 extends IGenerator {
7
8    public CharSequence doGenerateS(Resource input);
9
10 }
```