UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PROPOSTA E VALIDAÇÃO DE SOLUÇÃO DE COMPUTAÇÃO EM NUVEM PARA REDES COM DISPOSITIVOS NACIONAIS

ETTORE ZUGLIANI

ORIENTADOR: PROF. DR. CESAR AUGUSTO CAVALHEIRO MARCONDES

São Carlos – SP

Fevereiro/2016

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

PROPOSTA E VALIDAÇÃO DE SOLUÇÃO DE COMPUTAÇÃO EM NUVEM PARA REDES COM DISPOSITIVOS NACIONAIS

ETTORE ZUGLIANI

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Redes de computadores

Orientador: Prof. Dr. Cesar Augusto Cavalheiro Marcondes

São Carlos – SP

Fevereiro/2016

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária UFSCar
Processamento Técnico
com os dados fornecidos pelo(a) autor(a)

Z94p

Zugliani, Ettore
Proposta e validação de solução de computação em nuvem para redes com dispositivos nacionais / Ettore Zugliani. -- São Carlos : UFSCar, 2016.
83 p.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2016.

1. Redes. 2. Computação em nuvem. 3. Virtualização. 4. OpenStack. 5. Neutron. I. Título.

UNIVERSIDADE FEDERAL DE SÀO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Ettore Zugliani, realizada em 26/02/2016:

Prof. Dr. Cesar Augusto Cavalheiro Marcondes
UFSCar

Prof. Dr. Luis Carlos Trevelin
UFSCar

Prof. Dr. Eleri Cardoso
UNICAMP

AGRADECIMENTOS

Agradeço a todos que me apoiaram direta e indiretamente, mas principalmente aos aqui mencionados.

Agradeço aos meus pais Pedro e Neide, bem como minha irmã Ana Luiza e minha família por me trazerem até aqui, me incentivarem nessa jornada e estarem sempre presentes.

Agradeço também a minha noiva Mariane que sempre esteve comigo durante toda esta jornada, desde a graduação até o final do presente trabalho, me apoiando, dando forças e muito afeto até nos momentos mais complicados.

Agradeço a todos os amigos que o Departamento de Computação proporcionou são tantos nomes que não caberia citar todos aqui, mas vocês todos também me ajudaram e estiveram comigo durante diversas etapas dessa jornada.

Agradeço por fim a todos os professores do Departamento de Computação da UFSCar, pois sem vocês eu claramente não teria o conhecimento necessário para trilhar esse caminho. Em especial agradeço ao professor Cesar A.C. Marcondes que me ensinou muito, proporcionou diversas oportunidades e experiências.

Enfim a todos vocês muito obrigado por tudo.

Contos de fadas são a pura verdade: não porque nos contam que os dragões existem, mas porque nos contam que dragões podem ser vencidos

Neil Gaiman, Coraline

RESUMO

Computação em nuvem e virtualização são assuntos recorrentes utilizados em uma variedade de sistemas a fim de prover soluções eficazes, escaláveis, de facilitada manutenção e de menor custo. Dentre os principais nomes da área se destaca o *software OpenStack* que provê virtualizações em diversos níveis, desde máquinas virtuais até redes virtuais. O *OpenStack* possui código aberto e seu módulo de redes *Neutron* conta com o suporte a diversos grandes fabricantes como Cisco, Brocade e Arista, no entanto, até o presente momento não há suporte para nenhum fabricante nacional. Este trabalho propõe a construção de uma solução nacional para virtualização de redes através da construção de um *driver* para o módulo de redes do *OpenStack*, que possa suportar os equipamentos da fabricante brasileira Datacom. O texto apresenta a principio o cenário atual de computação em nuvem para redes, além de, um estudo sobre o *OpenStack* e seu módulo de redes *Neutron*, portanto, tomando esse estudo como ponto de partida são apresentados diversos diagramas a fim de propor uma solução. Esta solução é então construída utilizando a linguagem *Python* e boas práticas de programação alcançando ao final uma proposta sólida e altamente modular, por fim, esta proposta é validada por meio de testes unitários que são inclusive necessários para a aprovação do código na comunidade. O trabalho resulta em um ambiente *OpenStack* funcional utilizando os servidores da UFSCar que se comunica de maneira satisfatória com equipamentos nacionais de redes, além contribuir com o desenvolvimento do *OpenStack*.

Palavras-chave: Redes, Computação em nuvem, Virtualização, *OpenStack*, *Neutron*, *Python*

ABSTRACT

Cloud and virtualization are both recurring subjects, today they are used in a variety of systems in oder to provide solutions that are effective, scalable, with easier maintenance and les costly. Among the main names of the area we have the OpenStack software which provides various levels of virtualization from virtual machines to virtual networks, OpeStack code is open source and it‘s network module, Neutron, has the support of several manufactors like Cisco, Brocade and Arista but untill now there is no support for a national manufactor. This work proposes the construction of a national solution for network virtualization, proposing the construction of a driver for the OpenStack network module that supports the equipments of the brazilian manufactor Datacom. This text features first a overview of the actual environment of cloud computing, a study about OpenStack and it‘s network module Neutron, and them taking this study as starting point a set of diagrams are presented in order to propose a solution. This solution is them built using the programing language Python and good practices of programing and them at the end of te implementation presents a solid and highly modular solution. Therefore this proposal is validated through a series of unity tests which are necessary for the approval of the code by the comunity. This work results in a working OpenStack environment using the UFSCar servers, which comunicates in a satisfactory manner with national network equipments, besides the contribution with OpenStack development.

Keywords: Networks, Cloud computing, Virtualization, ,OpenStack ,Neutron ,Python

LISTA DE FIGURAS

LISTA DE TABELAS

LISTAGENS

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Computação em nuvem é um assunto recorrente em computação e hoje é utilizada em uma variedade de sistemas. O uso de nuvens e virtualizações prove diversas vantagens, entre elas menor custo, manutenção facilitada, maior escalabilidade e maior heterogeneidade. Dentre os softwares provedores de nuvem se obtém soluções abertas ou proprietárias, dentre as soluções abertas uma das mais utilizadas é o *OpenStack*.

OpenStack provê uma plataforma aberta para computação em nuvem, pública ou não. Além disso, possui diversos recursos, é escalável, de fácil extensão e altamente modular. O projeto é aberto sob a licença *Apache 2.0*. Devido a sua modularidade é possível o desenvolvimento para uma área do *OpenStack* sem ter o conhecimento de outra. Por exemplo, é possível modificar o funcionamento da virtualização sem que a conectividade sofra mudanças. Neste trabalho o módulo mais importante será o *Neutron*, responsável pela conectividade e rede do *OpenStack*.

Comparado a outros módulos do *OpenStack* o *Neutron* é recente, possuindo cerca de um ano. A principal função deste módulo é prover "redes como serviço"para os outros módulos do *OpenStack*, sendo portanto, de alta importância pois, do contrário não haveria comunicação confiável em uma nuvem *OpenStack*.

Para que seja possível configurar um determinado dispositivo de rede a partir da interface do *OpenStack* (que também é um módulo), é necessário um *plugin* que faça a comunicação entre o Neutron e o aparelho físico. Este *plugin* difere de acordo com cada tecnologia, podendo variar em tipo de comunicação (local, VLAN, GRE, etc) e *hardware* de fabricantes (Datacom, Cisco, Brocade, etc).

Um dos maiores avanços do *Neutron* é a criação do *plugin Modular Layer 2* (ML2), que facilita a integração de diferentes tecnologias de redes. Entretanto, no Neutron é possível utilizar apenas um *plugin* por vez em uma determinada rede, impossibilitando a comunicação entre

dispositivos de rede de fabricantes diferentes, já que cada fabricante utilizaria seu próprio *plugin*. O *plugin* ML2 provê uma camada onde essas tecnologias podem se comunicar utilizando *drivers* específicos que por sua vez se conectam com o *plugin* ML2, e este então, faz a conexão com o Neutron. Neste caso, cada fabricante constrói um driver que se conectará a um *plugin* em comum (ML2).

O intuito original deste projeto é a criação de um *plugin* para o suporte de redes envolvendo os equipamentos nacionais da Datacom. No entanto, isto inviabiliza o uso do equipamento da Datacom junto com os de outros fabricantes. Alcançando essa constatação uma outra solução foi a utilização do *plugin* ML2 e a construção de um *driver* para este. Dessa forma, possui-se o suporte apenas para as operações comuns das tecnologias, porém, com o uso de diversos dispositivos de redes de diferentes fabricantes na mesma rede.

De maneira a facilitar a leitura, esse trabalho foi dividido em sete capítulos além deste (introdução). O capítulo 2 apresenta uma visão geral sobre tecnologias de computação em nuvem, apresentando o *OpenStack* e seus concorrentes, e por fim os trabalhos similares a este. Já o capitulo 3 apresenta com mais detalhes a plataforma *OpenStack*, bem como o módulo *Neutron*, o uso de um *plugin* e o uso de um *driver* ML2. O capitulo 4 apresenta a arquitetura planejada para o *plugin* e como esta foi adaptada para o *driver*, bem como as diferenças entre ambos.

O capítulo 5 detalha todos os módulos da implementação do *driver*, como foram estruturados e suas diferentes bibliotecas. Para o desenvolvimento e testes da implementação foi necessário um ambiente, o estudo do funcionamento e os passos para replicar este ambiente. Estas atividades são detalhadas no capítulo 6. O capitulo 7 apresenta os testes e validações produzidos a fim de demonstrar o funcionamento deste projeto. Por fim, o capítulo 8 apresenta as contribuições deste trabalho, bem como as adequações que podem ser feitas no posteriormente.

Capítulo 2

TECNOLOGIAS E TRABALHOS RELACIONADOS

Este capítulo tem como objetivo abordar a evolução dos *softwares* provedores de nuvem, como estes evoluíram de virtualizadores para soluções completas que hoje englobam também a virtualização de redes. Em seguida são apresentados alguns exemplos de *softwares* tanto abertos quanto proprietários. Por fim são apresentados alguns trabalhos similares a este para que o leitor tenha uma melhor compreensão do escopo deste trabalho.

2.1 Computação em Nuvem para Redes

O conceito de computação em nuvem começou a ser desenvolvido usando a virtualização de sistemas como base. Ao longo do tempo esse conceito começou a incorporar novos vocabulários, como computação distribuída, serviços de *software* e mais recentemente englobou também a área de redes. Com toda essa infraestrutura hoje é possível prover nuvens bem estruturadas, com menos *overhead*, menos custo, mais segurança e com mais facilidades para o usuário final.

A área de redes engloba várias frentes de um sistema de computação em nuvem, desde virtualização até a segurança e isolamento. Através da virtualização é possível que nós inteiros sejam emulados logicamente. Essa emulação ajuda na redução de preço e também reduz um possível *overhead* na transmissão de dados. No quesito de segurança é possível que uma rede seja isolada do ponto de vista do usuário e desta maneira, os dados são isolados e os usuários não tem acesso aos dados dos demais usuários. No quesito de elasticidade, utilizando uma rede virtualizada não tem-se as limitações físicas que uma rede pode impor sobre a elasticidade, como por exemplo a limitação de portas em um comutador. A monitoração também é simplificada quando a rede é virtualizada, pois dessa maneira a rede pode ser monitorada, ponta a ponta, a distância.

2.1.1 Desafios na área

Todas as vantagens de uma computação em nuvem vêm com uma sobrecarga de complexidade, ou seja quanto mais robustez e quanto mais funcionalidades uma nuvem carrega, mais difícil se torna a sua implementação e manutenção. Isso se deve a mistura de diversas tecnologias, onde as versões mais atuais possuem mais vantagens, porém as versões mais antigas se integram melhor com outras tecnologias utilizadas. Quando se trabalha em apenas uma área da computação, como Redes, a atualização é simples e muitas vezes os problemas só aparecem quando tem-se a junção com outras áreas.

Por esse motivo é muito importante manter a modularidade em serviços de provedores de nuvem, sem essa modularidade a integração de componentes seria difícil de alcançar. Isso também se estende para a área de redes. Um sistema bem modularizado irá requisitar que o módulo de redes faça o isolamento entre o tráfego de usuários, que, por sua vez, será configurado para faze-lo utilizando a tecnologia que a rede suporta.

Um exemplo de desafio nessa área é o uso de VLANs para a virtualização de redes, o conceito de VLANs é simples, um campo a mais nos pacotes de redes que provê isolamento de tráfego, permitindo várias redes lógicas dentro de uma rede física. No entanto, como mencionado previamente, a solução com VLANs escala muito rápido, o que demanda planejamento, para prevenir a criação de *loops* na rede. Outro problema é com a segurança. Como são amplamente utilizadas as VLANs também são alvos de ataques, que podem quebrar a segurança que estas oferecem.

2.1.2 Softwares para computação em nuvem

Esta seção apresenta alguns *softwares* para a implementação de computação em nuvem. Enquanto que alguns softwares são para uso publico (como por exemplo o *CloudStack*, o *OpenStack* e o *Eucalyptus*) outros são apenas para uso interno da empresa que os implementa, sendo o mais famoso e mais utilizado o *Amazon Web Services*. Também na categoria de *softwares* privados tem-se o *Google Cloud Platform*, o *CloudBees, vCloud*, entre outros.

Eucalyptus (HPE..., 2016) é um software gratuito e aberto para construir ambientes de nuvem que sejam compatíveis com os da Amazon *Amazon Web Services*. O *Eucalyptus* possui um acordo com a Amazon para garantir que seus serviços sejam compatíveis entre si. Esse sistema é escrito principalmente na linguagem Java, e provê um robusto e configurável sistema que permite inclusive transferir conteúdo do *Eucalyptus* para o *Amazon Web services* através de sua interface.

CloudStack (APACHE..., 2015) é também um *software* gratuito, seu foco é em utilizar diversos hipervisores existentes para a criação de máquinas virtuais, dentre eles KVM, VMWare e *XenServer*. Também possui controle de redes, possuindo roteadores e *firewalls* virtuais.

Por fim tem-se o *software OpenStack* (HOME..., 2016) assim como os outros apresentados possui desenvolvimento aberto é desenvolvido utilizando a linguagem *Python*, pode ser controlado através de uma interface REST e possui diversos módulos e possui uma vasta comunidade e desenvolvimento contínuo, inclusive com encontros anuais de desenvolvedores e usuários.

2.2 Trabalhos relacionados

Dentro da metodologia de desenvolvimento, foi realizado um estudo dos guias oficiais da página do Neutron (`https://wiki.openstack.org/wiki/Neutron`), de modo a realizar a instalação de um ambiente *OpenStack* em um servidor e a montagem de um ambiente de desenvolvimento propriamente dito, bom como a montagem do projeto em si Em seguida são apresentados os elementos considerados principais do Neutron de acordo com o estudo, junto de outros trabalhos científicos da área.

2.2.1 Drivers do Neutron

O principal material para a construção desse projeto são os próprios *drivers* já existentes que podem ser encontrados em (NEUTRON/NEUTRON/PLUGINS/ML2/DRIVERS..., 2015). Dentre os *drivers* já existentes os mais interessantes são o Cisco o Brocade e o Arista.

O *Driver* da Cisco, por possuir equipamentos de diversas arquiteturas suportados, possui metaclasses para definir uma base destes equipamentos. Construído dessa maneira as metaclasses ficam mais complexas, enquanto que os próprios *drivers* ficam mais simples.

Já o Arista possui apenas um driver que é dividido em dois módulos principais, o que realiza a comunicação com o *OpenStack* e o banco de dados. Desta maneira a codificação é simples, mas acaba por prejudicar a longevidade do driver, já que toda a inteligência está em apenas um módulo.

Por ultimo tem-se o *driver* da Brocade que possui uma lógica diferente. Como o fabricante por já possui um *plugin* que cobre as operações básicas, seu *driver* apenas faz chamadas para as operações já abrangidas pelo *plugin*, realizando apenas pequenas alterações.

2.2.2 Medidas de desempenho de redes

O trabalho de Mogul e Popa (2012) toma como partida o fato de que *cloud data centers* precisam de redes de alta performance para conectar os servidores entre si e estes com a rede externa. Enquanto que os provedores desses serviços proveem diferentes tamanhos e possibilidades de armazenamento, memória e máquinas virtuais apenas uma conexão básica e sem garantias em termos de redes é oferecida.

No entanto há uma divergência nessa questão. Enquanto que alguns consumidores gostariam de possuir mais garantias em relação a rede e os provedores gostariam de oferecer e principalmente cobrar por esses serviços, não há um acordo sobre como a performance da rede pode ser medida e como pode ser a cobrança sobre esta.

O artigo discorre portanto sobre esse assunto e propõe algumas garantias e medidas de performance. Alguns serviços oferecem uma rede exclusiva para cada usuário Essa solução, no entanto, é pouco escalável e pode ser muito cara. Outra maneira é atribuir um peso para a quantidade de trafego de um usuário e utilizar este peso para alocar o trafego entre diferentes redes. Ainda outra possibilidade é oferecer serviços de redes diferenciados para cada topologia e deixar o usuário escolher qual das topologias é melhor para o seu serviço. Os autores basicamente dividem as possíveis soluções entre as que utilizam soluções físicas ou soluções baseadas no tempo.

2.2.3 Extendendo a virtualização de redes

O artigo de Benson (2011) também parte do princípio de que a computação em nuvem é amplamente utilizada e muito importante para o funcionamento de diversos serviços na atualidade. No entanto é salientado que vários provedores possuem deficiências quanto a segurança, privacidade, imprevisibilidade, confiabilidade e cumprimento de normas.

Segundo o autor estes problemas são oriundos da falta de controle que o usuário tem sobre a configuração das redes utilizadas ou sobre o sistema no geral. Este recebe uma configuração básica, que visa apenas prover a conectividade com suas máquinas virtuais deixando de lado funções como o controle de *firewall*, configurações de isolamento de redes, roteamento baseado em políticas de rede, controle sobre o endereçamento e acesso a otimizações (acelerações de protocolo, controle de encaminhamento, cache distribuído, etc).

O artigo portanto apresenta um *framework* diferenciado, chamado de *CloudNaas*, este além de prover serviços de máquinas virtuais e armazenamento, também possui serviços de redes

dando ao usuário acesso a diversas virtualizações de redes, permitindo que o usuário utilize diversas aplicações, estas por sua vez podem prover controle sobre diversos aspectos da rede. O artigo então detalha como foi concebido tal *framework*, e as diversas tecnologias utilizadas no mesmo.

2.2.4 OpenStack em conjunto com NFV

O texto do autor Ge (2014) começa apresentando a ideia de que *middleboxes* são utilizados para prover diversas funcionalidades em redes modernas, entre elas segurança através de *firewalls*, performance através de aceleradores otimizados e qualidade de serviço através de *deep packet inspection*. NFV é uma tecnologia que atua nessa área implantando diversas funcionalidades utilizando *software* no lugar de *hardware*.

No entanto, para algumas soluções o uso de apenas NFV trás um impacto na performance, como por exemplo em *deep packet inspection* e *network adress translation*. Em casos como esses ainda é necessário o uso de *hardwares* específicos, como por exemplo FPGAs. O objetivo desse trabalho é eliminar essa restrição desenvolvendo um *framework, OpenANFV* que suporte a virtualização desses *hardwares*.

O *OpenANFV* possui três principais funcionalidades. Primeiramente possui um gerenciamento automatizado, propondo um provisionamento automatizado para multiplas instâncias. O *framework* também possui elasticidade, permitindo que instancias sejam criadas, movidas e removidas em tempo real. Por último o *OpenANFV* possui integração com o *OpenStack*, funcionando como se fosse um de seus módulos.

Capítulo 3

OPENSTACK

Tendo explorado o tema de *softwares* para virtualização, e conhecendo algumas das alternativas, este capítulo aborda a principal tecnologia utilizada neste trabalho, *OpenStack*. A justificativa para a escolha dessa plataforma se deve principalmente à ampla base de usuários e desenvolvedores. No que segue será apresentado detalhes dessa plataforma, de modo que o leitor possa entender como a modularização funciona, e quais as características principais que estão presentes.

Dessa forma, o *OpenStack* assim como dito na seção 2.1.2, é uma das mais atrativas tecnologias para provisionamento de redes em nuvem. É o resultado de uma colaboração comunitária para a criação de uma plataforma aberta para computação em nuvem, seja esta nuvem pública ou privada. O projeto *OpenStack* visa prover soluções para diversos tipos de nuvens, possuindo as seguintes características principais: diversos recursos, escalável e de fácil implementação, constituído de uma série de módulos para diferentes soluções de nuvem. Todos os códigos do *OpenStack* estão disponíveis sob a licença Apache 2.0, de modo que qualquer pessoa possa submeter mudanças ao projeto. A figura 3.1 apresenta um exemplo da interface do OpenStack

Uma de suas principais características que provê robustez ao *OpenStack* e ao mesmo tempo facilita o desenvolvimento aberto da ferramenta é a sua modularidade. Esta será explora na próxima seção.

3.1 Estrutura modular do OpenStack

Pode-se dividir o *OpenStack* em 10 componentes principais, os quais são apresentados na Tabela 3.1. Dentre estes não é dado muito foco ao *Ceilometer, Heat e Trove*, pois estes ainda

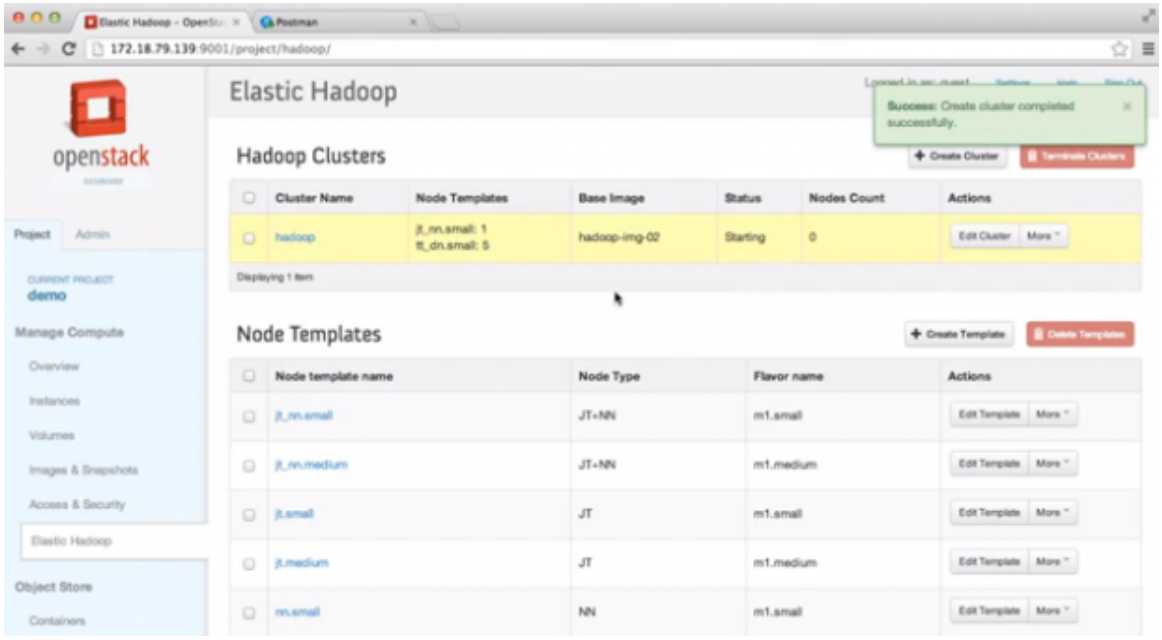

Figura 3.1: Dashboard do OpenStack, provido pelo módulo Horizon, tirado de (PROJECT..., 2013)

estão em constante desenvolvimento. No entanto os demais módulos possuem um interesse e importância maior na comunidade e na estrutura do *OpenStack*. Dentro de cada módulo tem-se outros componentes menores, enfatizando novamente a alta modularidade deste projeto.

A figura 3.2 apresenta uma visualização de como os módulos apresentados na Tabela 3.1 se comunicam. O usuário realiza as operações no *dashboard*, os componentes que permitem interface com o usuário conversam com o *dashboard* para tal, o principal módulo responsável pelo *dashboard* é o *Horizon*. O principal e mais complexo módulo do *OpenStack* é o *compute* que cuida do gerenciamento do ambiente virtual, armazenando as imagens das VMs no *image service* que por sua vez armazena os arquivos de disco no *object storage*. O *Networking* é responsável por prover conexão de rede ao *Compute*, assim como o *block storage* provê armazenamento persistente ao mesmo. Por fim, o *identity service* provê autorização e registro para os demais componentes.

3.2 Neutron

O Neutron, como pode ser visto em (OPENSTACK..., b) é um módulo novo, quando comparado com os outros existentes. Ele substitui o antigo módulo de redes, chamado de Quantum. Essa substituição ocorreu em 2013 com o lançamento do *OpenStack* versão Havana. A principal função do Neutron é prover "redes como serviço" (*network as a service*) para os dispositivos controlados por outros módulos do OpenStack, tornando-se um dos grandes núcleos do mesmo.

Módulo	Características
Nova	OpenStack Compute Construído para gerenciar e automatizar conjuntos de recursos computacionais. Trabalha com várias tecnologias de virtualização; kvm e xen podem ser utilizados como hipervisores
Swift	OpenStack Object Storage sistema escalável e redundante de armazenamento; objetos e arquivos escritos em múltiplos discos espalhados no data center.
Cinder	Provê armazenamento persistente e block-level para o uso de instâncias do Nova
Neutron	OpenStack Networking Antigo Quantum Sistema para gerenciar redes e endereços ip; Garante que a rede não será gargalo na implantação da rede.
Horizon	OpenStack Dashboard Interface gráfica para acesso, fornecimento e automação de recursos.
Keystone	OpenStack Identity Diretório central dos usuários, mapeados com serviços Openstack que podem acessar.
Glance	OpenStack Image Service Provê serviços de descoberta, registro e entrega de discos e imagens; Estas imagens podem ser usadas como templates; Store de backups.
Ceilometer	OpenStack Telemetry Service Centro de sistema de pagamento, provê contadores necessários para estabelecer taxas entre componentes.
Heat	Orchestration Gerencia múltiplas aplicações utilizando templates.
Trove	Database Provedor de database as a service. Prove bancos de dados relacionais ou não.

Tabela 3.1: Módulos do OpenStack.

Por exemplo, as VMs controladas pelo módulo Nova podem criar/controlar uma rede utilizando-se dos serviços do Neutron. Deste modo é possivel que usuários (também chamado de *tenant*) configurem políticas e construam serviços avançados de redes, além de possuir uma interface gráfica para realizar tais configurações.

O plugin *Neutron* se utiliza de três conceitos básicos para dados: *Networks*, *Subnetworks* e *Ports*, que serão explicados a seguir.

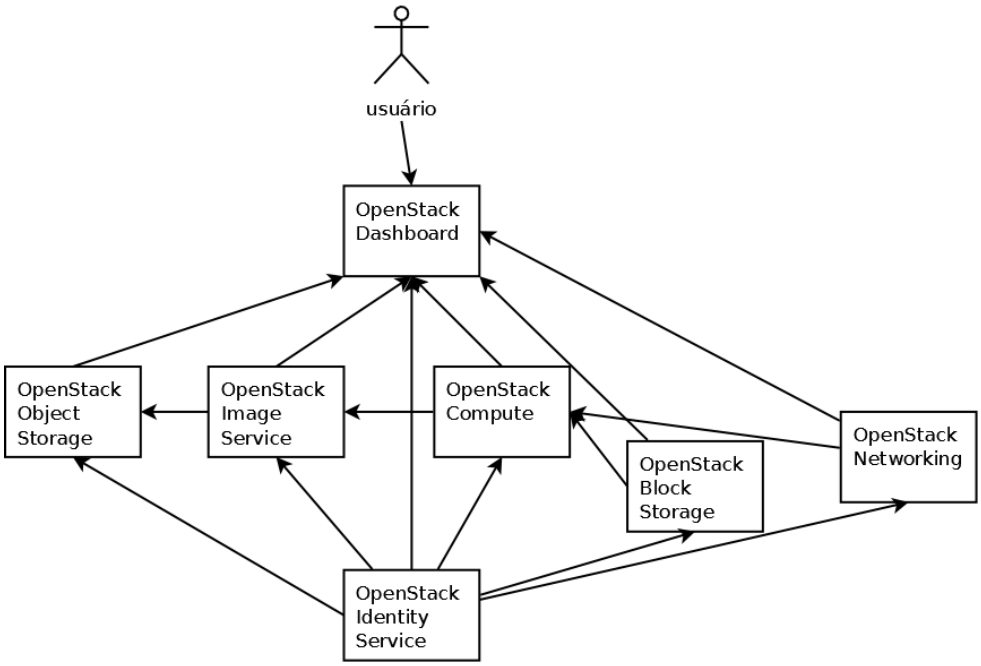

Figura 3.2: Estrutura básica do OpenStack

1. ***Network***: Uma *network* (ou rede) é um domínio de *broadcast* virtual de *layer-2* isolado de outros domínios. Este domínio é reservado por um usuário após a criação da rede, até que a esta seja configurada para permitir compartilhamento de dados. Existe um número máximo de *networks* que um usuário pode criar.

2. ***Subnet***: Uma *subnet* (sub-rede) representa um bloco de endereços *IP* que podem ser usados para atribuição de endereços à instâncias virtuais. Cada sub-rede precisa de um CIDR (*Classless Inter-Domain Routing*) e precisa estar associado a uma rede. Endereços *IP*s podem ser selecionados de uma CIDR de sub-rede ou de uma *pool* que podem ser especificadas pelo usuário. A sub-rede, opcionalmente, pode ter um *gateway*, uma lista de *DNS name servers*, e *host routes*. Essa informação é enviada para instâncias cujas interfaces estão associadas com a sub-rede.

3. ***Port***: Uma *port* (porta) representa uma porta de um *switch* virtual em um *switch* de uma rede lógica. Instâncias virtuais conectam suas interfaces nas portas. A porta lógica também define o endereço *MAC* e os endereços *IP* a serem atribuído às interfaces conectadas a ela. Se endereços *IP* são associados a uma porta, isso implica que a porta é associada com uma sub-rede, e o endereço *IP* foi recuperado de uma *pool* de alocação para uma sub-rede específica.

O *Neutron* suporta as seguintes operações básicas sobre cada tipo de dado do Neutron (*Network* , *Subnetwork* e *Port*), que são:

- ***Create***: Cria uma instância do tipo de dado requerido, passados os parâmetros necessários;
- ***Show***: Recupera os dados de uma instância do tipo de dado requerido;
- ***Update***: Altera os dados de uma instância do tipo de dado requerido;
- ***Delete***: Remove uma instância do tipo de dado requerido.

3.2.1 Arquitetura do Neutron

Devido a proposta de alta modularidade do *OpenStack* o módulo *Neutron* também é composto de diversos módulos como pode ser observado pela Figura 3.3 tendo como principal o *neutron-server*. Nesta figura pode-se observar os seguintes componentes:

1. ***neutron-server***: Componente principal, no qual se encontra a conexão com o *plugin*, banco de dados, fila de mensagens e comunicações externas, sendo basicamente composto de três subcomponentes:

   (a) ***API REST***: Normalmente pode ser acessada pela porta 9696. Responsável por expor os recursos lógicos oferecidos pelo módulo *Neutron* tais como *network, subnet e port*. Também é responsável por serializar respostas e requisições do *Neutron*.

   (b) ***serviço RPC***: Componente responsável pela comunicação com os agentes. Facilita a comunicação com o *OpenStack*, é de fácil configuração e provê comunicação bidirecional com os agentes. O uso deste componente é opcional, dependendo do *plugin* utilizar ou não agentes.

   (c) ***plugin***: Componente principal do *neutron-server* e portanto do *Neutron*, possui a lógica do módulo já que os outros componentes do *neutron-server* são, basicamente, utilizados para comunicação. Apenas um *plugin* pode estar ativo em dado momento. Este *plugin* pode ser implementado da maneira que o desenvolvedor preferir, podendo possuir extensões diversas, banco de dados próprio, entre outras características. No *OpenStack o plugin* a ser utilizado é definido na variável *core plugin*.

   Na prática, todos os *plugins* são uma extensão da classe *NeutronPluginBaseV2* (V2 vem da segunda versão do *Neutron*, que utiliza *subnet*). Esta possui diversos métodos abstratos que devem ser implementados no *plugin*, sendo que alguns não são necessários, porém são serviços oferecidos pelo *Neutron*. A classe *NeutronDbPluginV2* é comummente utilizada por desenvolvedores, já que possui um *framework* para manipulação de banco de dados.

2. ***banco de dados***: Referido neste trabalho como banco de dados do Neutron (para diferenciar do banco de dados do plugin ou do driver, mais detalhes na seção 3.2.2), possui dados necessários para o funcionamento do Neutron. Por exemplo a relação entre *networks, subnets* e *vlans*. Este banco de dados deve ser acessado e atualizado pelo plugin, sendo independente de qualquer outro banco de dados que o plugin possa adicionalmente utilizar.

3. ***fila de mensagens***: simplesmente um *buffer* de mensagens para que haja a serialização das mesmas. Importante para manter a boa comunicação entre o *neutron-server* e os agentes.

4. ***agente L2***: Assim como todos os agentes a seguir este é opcional. O uso ou não de um agente L2 irá depender da implementação do *plugin*. Normalmente funciona sobre um *hypervisor*, notificando quando um dispositivo é adicionado ou removido, e conectando logicamente novos dispositivos.

5. ***agente L3***: Funciona em um nó da rede. Pode ser um ponto único de falha se for utilizado apenas um (podem ser utilizados vários agentes do mesmo tipo). Tipicamente utiliza *namespace* para separação de usuários.

6. ***agente DHCP***: Assim como o agente L3 normalmente fica em um nó da rede, e também utiliza *namespaces*.

7. ***outros agentes***: Os agentes listados anteriormente são os mais comuns. Exemplos de outros agente são: agente *OVS (Open vSwitch), firewall, metadata*, balanceamento de carga, entre outros.

3.2.2 Armazenamento de dados

Como mostrado na seção 3.2.1 os *plugins* para *Neutron* normalmente fazem uso de dois bancos de dados, os quais servem propósitos diferentes:

1. **Banco de dados do *Neutron***: contém dados referentes ao próprio *Neutron*, por exemplo a associação de usuários com redes.

2. **Banco de dados do *plugin***: este contém dados internos do *plugin*, que variam para cada vendedor (fabricante do comutador). Por exemplo, se na criação de uma rede são utilizadas *VLANs* podem ser guardados dados das *VLANs*, e quais portas estão envolvidas.

A atualização do banco de dados pode ser feita de dois modos: assíncrono e síncrono, os quais são explicados a seguir.

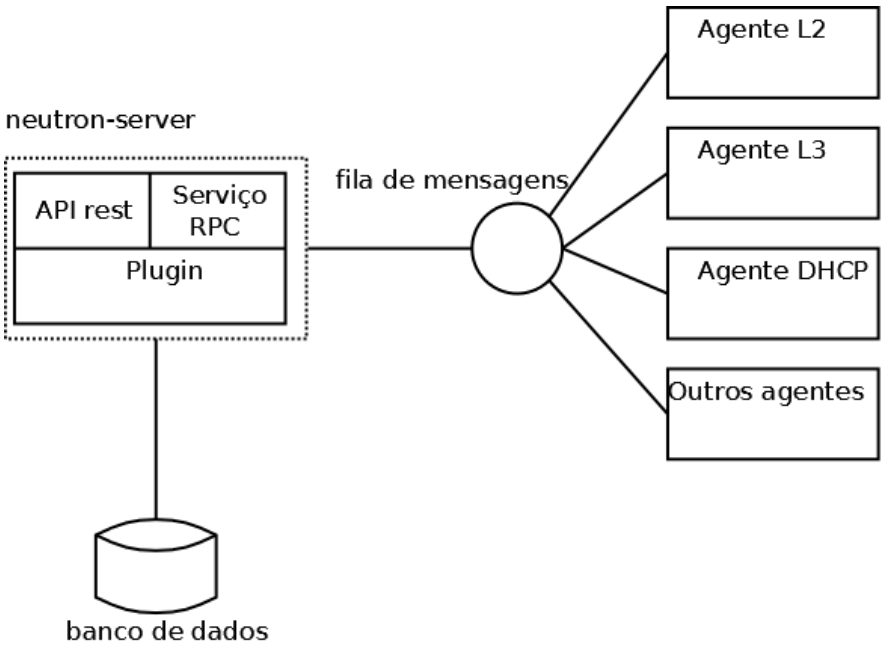

Figura 3.3: Estrutura do módulo *Neutron*

1. **Assíncrono**: A atualização do banco de dados ocorre independente da operação que desencadeou esta atualização. Por exemplo, ao criar uma rede o banco de dados é atualizado antes da rede de fato ser criada, como se a operação já tivesse retornado sucesso. Esta abordagem torna a atualização mais rápida e mais dinâmica do ponto de vista do usuário, além da implementação ser mais simples, porém trata-se uma operação com chances de causar erro.

2. **Síncrono**: Nesta abordagem o banco de dados é atualizado apenas depois que a operação é bem sucedida, desta maneira o banco de dados é mais consistente e menos suscetível a erros, porém as atualizações podem tornar mais lenta a experiencia do usuário.

3.2.3 *API do Neutron*

A API do *Neutron*, como pode ser visto em (OPENSTACK..., a) é uma interface que deve ser implementada para a criação de um *plugin*. Trata-se do módulo que traduz as operações em métodos programáveis, como por exemplo nas Operações CRUD (*create, read, update, delete*). Cada *plugin* deve implementar tais métodos, configurando seu próprio banco de dados, os Tipos de Dados da API e o *switch*.

Para compreender o fluxo de operações do módulo *Networking* do *Openstack*, é apresentado o seguinte caso: criar uma rede, associar uma sub-rede a essa rede, e inicializar uma máquina virtual que, por sua vez, está associada e essa rede. Para a deleção, é necessário apenas remover

a máquina virtual, remover qualquer porta associada com a rede, e por fim remover as redes utilizadas. O *Networking do OpenStack* lida com a deleção de quaisquer sub-redes associadas com redes deletadas.

1. ***Criar a rede***: O usuário cria uma rede, por exemplo, por exemplo a rede*net1* com id *net1_id*.

2. ***Associar a sub-rede com a rede***: O usuário associa uma sub-rede com essa rede recém criada. Por exemplo, o usuário associa a sub-rede 10.0.0.0/24 com a rede *net1*.

3. ***Inicializar a Máquina Virtual e associá-la a rede***: O usuário inicializa a máquina virtual (VM) e especifica um único NIC que conecta a uma rede.

   Os seguintes exemplos usam o módulo *Nova* para inicializar a VM.

   No primeiro exemplo, o *Nova* conecta com o *Networking* do *OpenStack* para criar o NIC e associá-lo com a rede *net1*, que possui o ID *net1_id*:

   ```
   $ nova boot < server\_name> --image <image> --flavor <flavor> --nic port-
       id=<port1\_id>
   ```

   No segundo exemplo, é necessário primeiro criar a porta *port1*, e então inicializar a máquina virtual com a porta específica. O *Networking* do *OpenStack* cria a NIC e a associa com a porta *port1*, que possui o ID *port1_id*:

   ```
   $ nova boot < server\_name> --image <image> --flavor <flavor> --nic port-
       id=<port1\_id>
   ```

   O módulo *Networking do OpenStack* escolhe e associa um endereço IP para a porta *port1* automaticamente.

4. ***Remover uma VM***: O usuário remove a Máquina Virtual.

   O *Nova* contacta o *Networking* do *OpenStack* e remove a porta *port1*.

   O endereço IP é retornado para uma *pool* de endereços IP disponíveis

5. ***Remover qualquer port***: Se o usuário criou alguma *port* e a associou com uma *network*, o usuário necessita deletar a *port*.

6. ***Remover a network***: O usuário deleta a rede. Essa operação deleta uma rede do *Networking do OpenStack* e as suas sub-redes associadas provisionadas que não tenham nenhuma porta configurada na rede.

3.3 Plugin ML2

O grande problema decorrente da utilização de *plugins* de fabricantes específicos é que o *OpenStack* carrega apenas um *plugin*. Considerando que os *plugins* implementavam toda a comunicação do *Neutron* com o ambiente físico, *plugins* de apenas um fabricante interagem com um *OpenStack*. Portanto, tendo um *plugin* para o *switch Datacom DM4100*, este não poderia ser utilizado em conjunto com comutadores de fabricantes diferentes.

A utilização do ML2 foi bem recebida pela comunidade, assim como os *Mechanism Drivers*. Desse modo, a arquitetura foca em trabalhar como um *Mechanism Driver* que se comunique, de maneira eficaz, com o equipamento Datacom, o DM4100.

O plugin ML2 possui dois componentes: *TypeDriver e MechanismDriver*. Um *TypeDriver* gerencia cada tipo de comunicação entre redes, que podem ser dos seguintes tipos: *local, vlan, gre, vxlan*. Desta maneira, o ML2 realiza a comunicação entre o *MechanismDriver* e o *OpenStack*, mantendo estados da rede que possam ser necessários e provendo validações da redes e alocação de usuários. É importante destacar que, mesmo sendo possível utilizar vários *MechanismDrivers* diferentes ao mesmo tempo, deve-se escolher apenas um *TypeDriver*.

Cada dispositivo de rede é gerenciado por um *MechanismDriver*, que é responsável por receber uma dada informação, que é estabelecida pelo *TypeDriver*, e validar a aplicação dessa informação em todos os mecanismos de redes habilitados, para que estes compreendam a informação. Portanto todo *MechanismDriver* precisa suportar o mesmo *TypeDriver* utilizado. No caso deste projeto o *TypeDriver* escolhido é *vlan*.

A interface do *MechanismDriver* suporta a criação, deleção e atualização de portas e recursos de redes. Para cada uma destas ações são executados dois métodos, *precommit* que é chamado dentro do contexto do banco de dados e o *postcommit*, chamado após a operação do banco de dados terminar. O *precommit* é utilizado pelo *MechanismDriver* para validar a ação sendo executada e fazer as mudanças necessárias ao banco de dados interno do driver, sendo que este método não consegue se comunicar com nada fora do *Neutron*. O método *postcommit* é responsável por encaminhar as mudanças executadas aos recursos responsáveis por executar estas mudanças.

No código do Neutron, os drivers já construídos para o ML2 podem ser encontrados em */opt/stack/neutron/neutron/plugins/ml2/drivers*. Para o estudo sobre as chamadas dos drivers, foram utilizados diversos deles, como por exemplo o dos fabricantes *Brocade, Cisco e Arista*. O maior problema encontrado é que quando já existe um *plugin* do mesmo fabricante, boa parte do código do *driver* utiliza trechos do plugin. Isto dificulta tanto a abstração do código quanto a

compreensão do que é necessário para o desenvolvimento de um driver.

Capítulo 4

PROPOSTA DE ARQUITETURA

Tendo estudado a arquitetura do *OpenStack* bem como a arquitetura do *Neutron*, as funções e implementação de um *plugin*, a mudança de foco de *plugin* para *driver* pela própria comunidade do *OpenStack* e sabendo como deve ser e se comportar um *driver*, tendo como base outros *drivers* já construídos.

Esta seção irá discorrer sobre o processo de construção da arquitetura do projeto. Para que fosse possível uma confecção da arquitetura de computação em nuvem para redes nacional, baseada nos *switches* da Datacom, foram analisados diversos *plugins* existentes do Neutron, apresentados na seção 2.2. A partir desse estudo, foram construídos dois casos de uso considerados elementares: o que segue a ação de criar uma rede (*Create Network*) e o de deletar uma rede (*Delete Network*). Em seguida é proposta uma arquitetura para o projeto.

Um processo similar é então apresentado para a construção do *driver*, no qual é levado em consideração principalmente a arquitetura já proposta para o *plugin*.

4.1 Plugin do Neutron

O *plugin* é o principal componente do módulo de Rede (*Networking*). É ele que se comunica com os equipamentos e traduz as mensagens de rede. Devido a modularidade e variabilidade do *OpenStack*, é comum que cada *plugin* siga uma implementação diferente. A repetição de alguns trechos de código faz com que novos módulos, classes e métodos sejam criados, facilitando o desenvolvimento. Um exemplo é a criação da classe *NeutronDbPluginV2*, derivada do fato de que muitos *plugins* utilizam um banco de dados externo. Outros exemplos de adições que facilitam são os diversos agentes e serviços que o *Neutron* provê. Esta facilidade para o desenvolvedor é fundamental para o crescimento do projeto, já que cada fabricante deve desenvolver seu próprio

plugin.

Exemplos de *plugins* existentes são o *Open vSwitch*, o qual é simples e possui apenas as operações básicas de CRUD, e pode ser utilizado até mesmo dentro de outros *plugins*. Tem-se também o *Linux bridge*, controladores *OpenFlow*, e *plugins* de fabricantes como a Cisco e Brocade, entre diversos outros. Para este estudo foram avaliadas a proposta de vários deles de modo a auxiliar na implementação e projeção deste trabalho.

Foi realizado o estudo de dois casos de uso presentes em outros *plugins*, o caso de uso do método de criação de uma rede *(Create Network)* e o caso de uso do método de remover uma rede *(Delete Network)*. O estudo desses casos de uso resultou em dois diagramas de sequência, um para cada caso de uso. Esses diagramas serão detalhados nos tópicos a seguir. É importante deixar claro que o ator nestes casos pode ser o próprio *OpenStack*, e que, como este utiliza *Network as a service* como provedor de serviços, uma requisição da API dos *plugins (create, delete, update, etc)* é atendida apenas quando for possível responder tal requisição. Portanto, requisições não são necessariamente executadas no momento que a chamada é repassada.

4.1.1 Caso de Uso Create Network

O ato de criar uma rede no *OpenStack* é equivalente a relacionar um usuário com uma *VLAN*. Deste modo, este usuário possui acesso a uma determinada rede, podendo então configurar a mesma.

Seguindo o fluxo do caso de uso da Figura 4.1, há uma requisição do usuário para criação de uma rede, esta requisição primeiramente é tratada pela classe *BrocadePluginV2*. Esta é a classe principal da requisição, a qual envia uma requisição para a classe *NeutronDbPluginV2* (classe pai da mesma). O banco de dados interno do *OpenStack* será atualizado contendo dados da rede que sejam relativos ao próprio *OpenStack*. Após o retorno do banco de dados, o *BrocadePluginV2* envia uma requisição para a classe *nosdriver*, que é responsável por se comunicar com os comutadores. Portanto, serão enviadas chamadas para os comutadores no formato que estes esperam *(Netconf)*, para que então possam configurar a nova rede. Após configuradas, a operação retorna para a classe *BrocadePluginV2*. No último passo, esta classe fará uma chamada a classe *models*, para que então seja atualizado o banco de dados interno do *plugin*, contendo informações internas da criação de rede. A operação então retorna sucesso ao usuário.

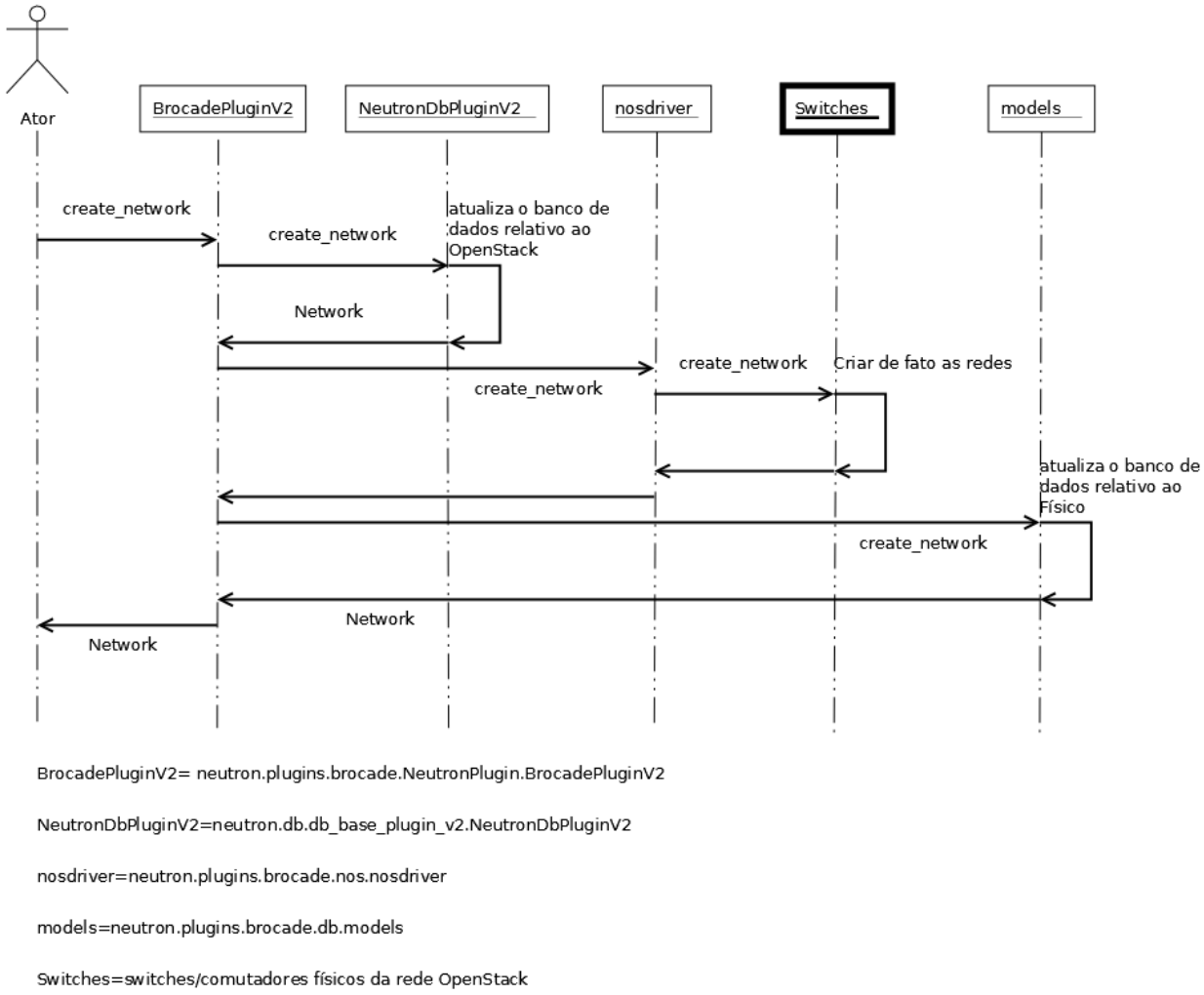

Figura 4.1: Caso de uso create network abstraido com base no plugin da Brocade

4.1.2 Caso de Uso Delete Network

Remover uma rede é um processo similar à criação de uma rede. Notar que no caso dos *plugins* analisados a atualização no banco de dados do *OpenStack* ocorre antes da deleção de fato ocorrer. Outra abordagem possível seria aguardar que o processo ocorra nos comutadores, e só então atualizar o banco de dados, o que neste caso é realizado com o banco de dados interno.

Seguindo o caso de uso da Figura 4.2, primeiro o usuário envia a requisição para remoção da rede. A classe *BrocadePluginV2* novamente é a classe principal. Esta envia uma requisição para a classe *NeutronDbPluginV2* que atualizará o banco de dados do *OpenStack*. A classe *BrocadePluginV2*, por sua vez, interage com a classe *models* para descobrir quais as portas do comutador fazem parte da rede que será removida. Isso é feito uma a uma por meio de uma repetição. Tendo todas as portas associadas a rede, é enviado para a classe *nosdriver*, o comando para deletar a rede, juntamente com os dados coletados, inclusive a *VLAN*. A *nosdriver* então

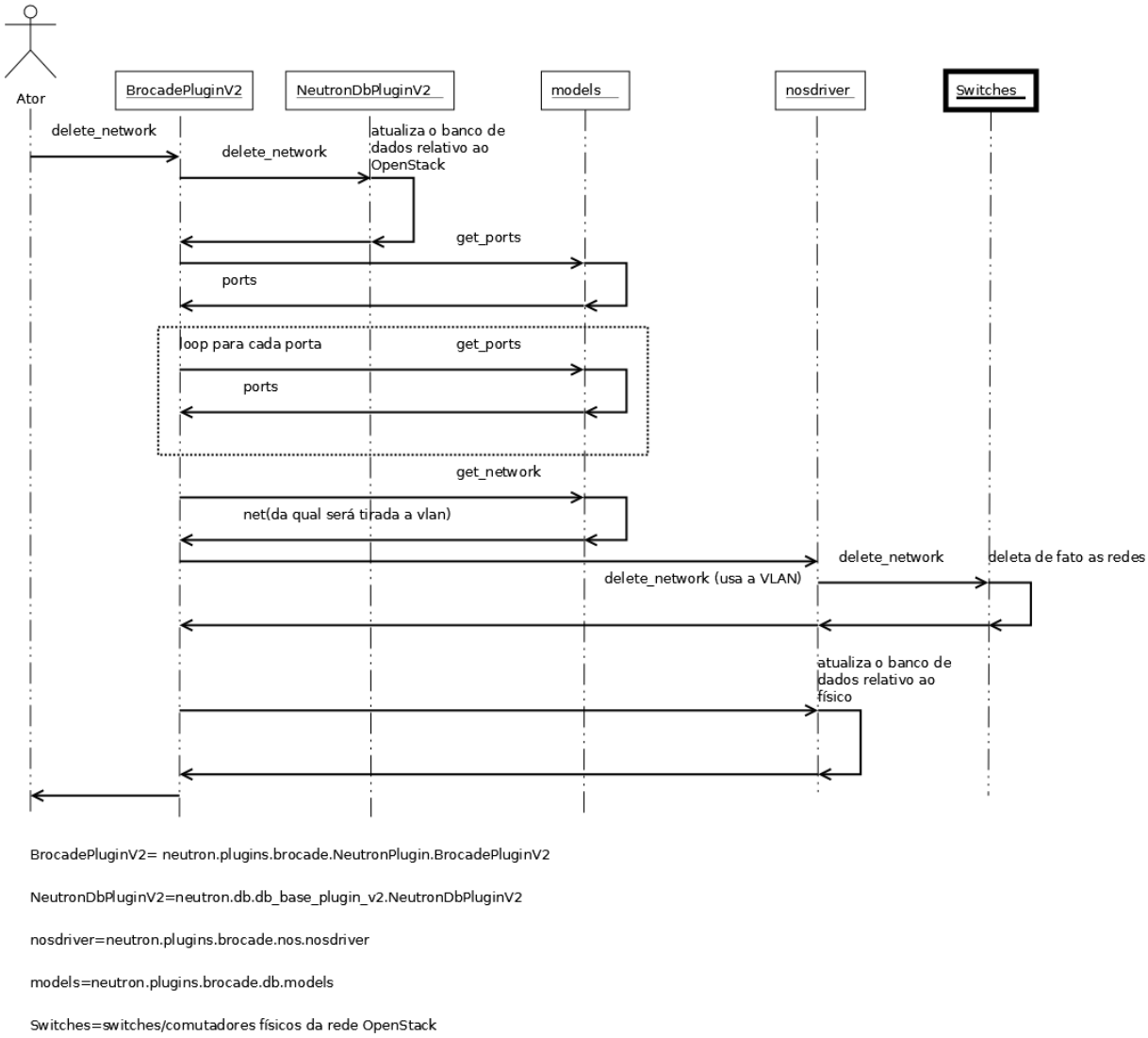

Figura 4.2: Caso de uso delete network utilizado no plugin da Brocade

exclui de fato a rede dos comutadores. Por fim, esta classe atualiza o banco de dados interno e a operação termina.

4.1.3 Arquitetura proposta do Plugin

Estudando principalmente os casos de uso apresentados, foi derivada uma arquitetura, a qual pode ser encontrada na figura 4.3 e possui 5 módulos, sendo estes descritos a seguir.

1. **main**: Classe principal do *plugin*, esta será a classe que implementa as operações básicas de CRUD (*Create, Read, Update* e *Delete*). Faz a interface com o *OpenStack*, e age como centro das operações.

2. **driver**: Classe que faz a interface com os comutadores físicos, traduzindo as mensagens

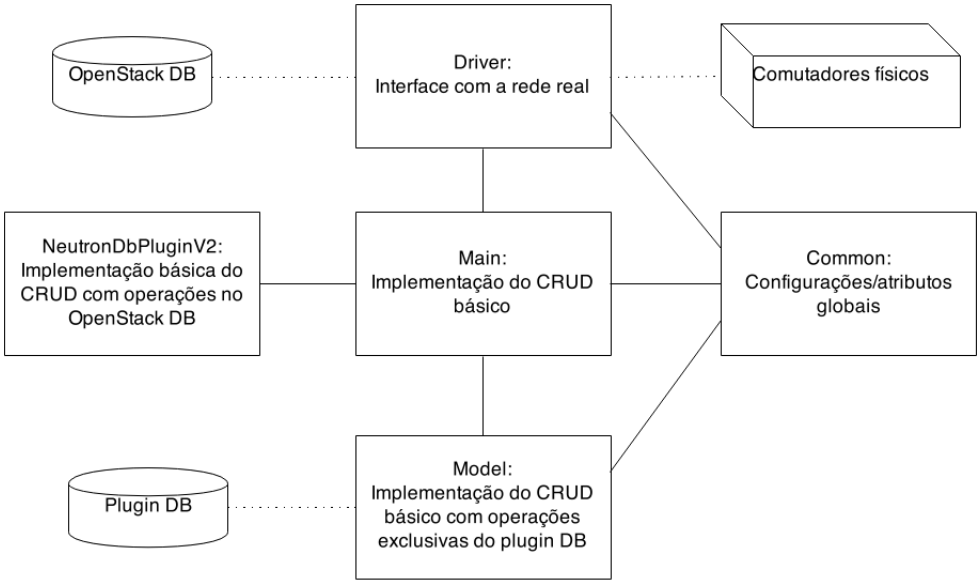

Figura 4.3: Arquitetura proposta para o plugin

da classe *main* para a linguagem que os comutadores processam.

3. **NeutronDbPluginV2**: Classe que se comunica com o banco de dados do *OpenStack*.

4. **Model**: Classe que se comunica com o banco de dados interno do plugin, possuindo dados específicos do plugin.

5. **Common**: Este módulo guarda atributos e configurações globais, e pode ser acessado pelos outros módulos.

4.1.4 Mudança para driver

A arquitetura descrita até agora expõe a implementação baseada em *plugin*, considerada promissora no escopo do projeto. Porém após um estudo mais aprofundado do *Neutron*, e de diversas sessões com os desenvolvedores foram expostas algumas desvantagens dessa abordagem. Uma delas é que o uso de *plugins* já está caindo em desuso na comunidade, mas principalmente expôs a impossibilidade de uma nuvem heterogênea, pois utilizando um plugin pode-se apenas usar comutadores de fabricantes que suportem uma tecnologia idêntica (normalmente apenas dentre os mesmos fabricantes). Desse modo, foi proposta uma nova arquitetura que pudesse solucionar as limitações quanto a heterogeneidade de fabricantes de comutadores em uma rede para a qual os próprios desenvolvedores aconselharam sobre o uso de *drivers* oposto ao uso de *plugins*. Esta nova arquitetura, que será apresentada a seguir, utiliza o já mencionado *plugin ML2*, focando no entanto na construção de um *driver* para este.

4.2 Driver do Plugin ML2

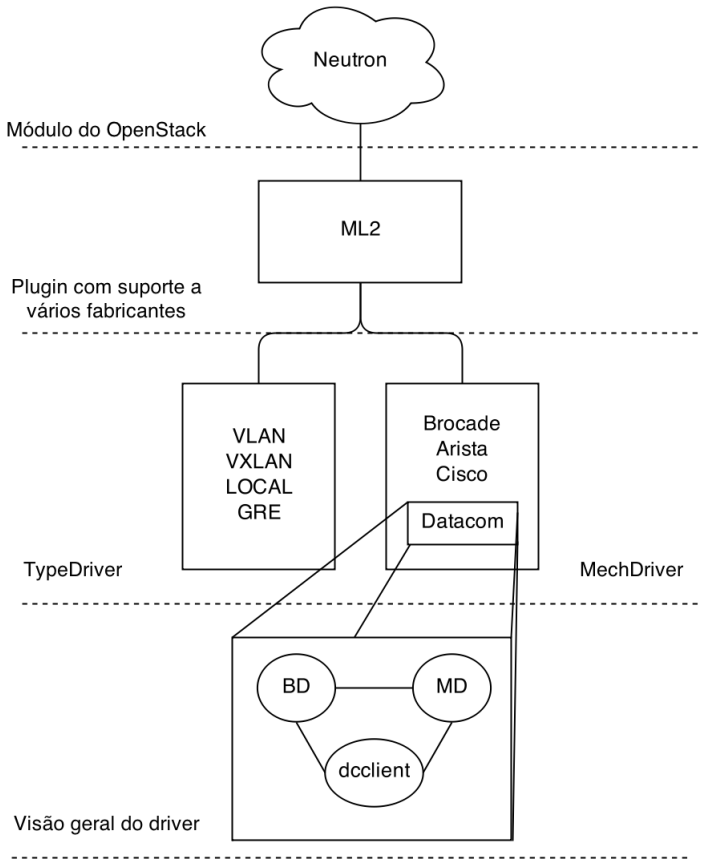

Figura 4.4: Ideia geral das camadas de abstação e dos componentes de um driver

A figura 4.4 apresenta a arquitetura geral do *driver* implementado e de como os componentes estão organizados. Nela pode-se ver que o *plugin* utilizado será o ML2, o qual possui dois módulos: *TypeDriver e MechDriver*, sendo que o *MechDriver* implementado possui três partes: banco de dados (BD), *MechDriver* (MD) e *dcclient*.

Cada dispositivo de rede é gerenciado por um *MechanismDriver*, que é responsável por receber uma dada informação (estabelecida pelo *TypeDriver*) e garantir a corretude da aplicação dessa informação em todos os mecanismos de redes habilitados, para que por fim estes compreendam a informação. Portanto, todo *MechanismDriver* precisa ter suporte ao mesmo *TypeDriver* utilizado. No caso do Projeto *DatacomMechanismDriver*, o *TypeDriver* escolhido é VLAN, já que o DM4100 (comutador utilizaco primeiramente) possui suporte a VLAN e os demais drivers desenvolvidos também possuem tal suporte. Tendo isso, conclui-se que o *DatacomMechanismDriver* será o objeto a ser desenvolvido.

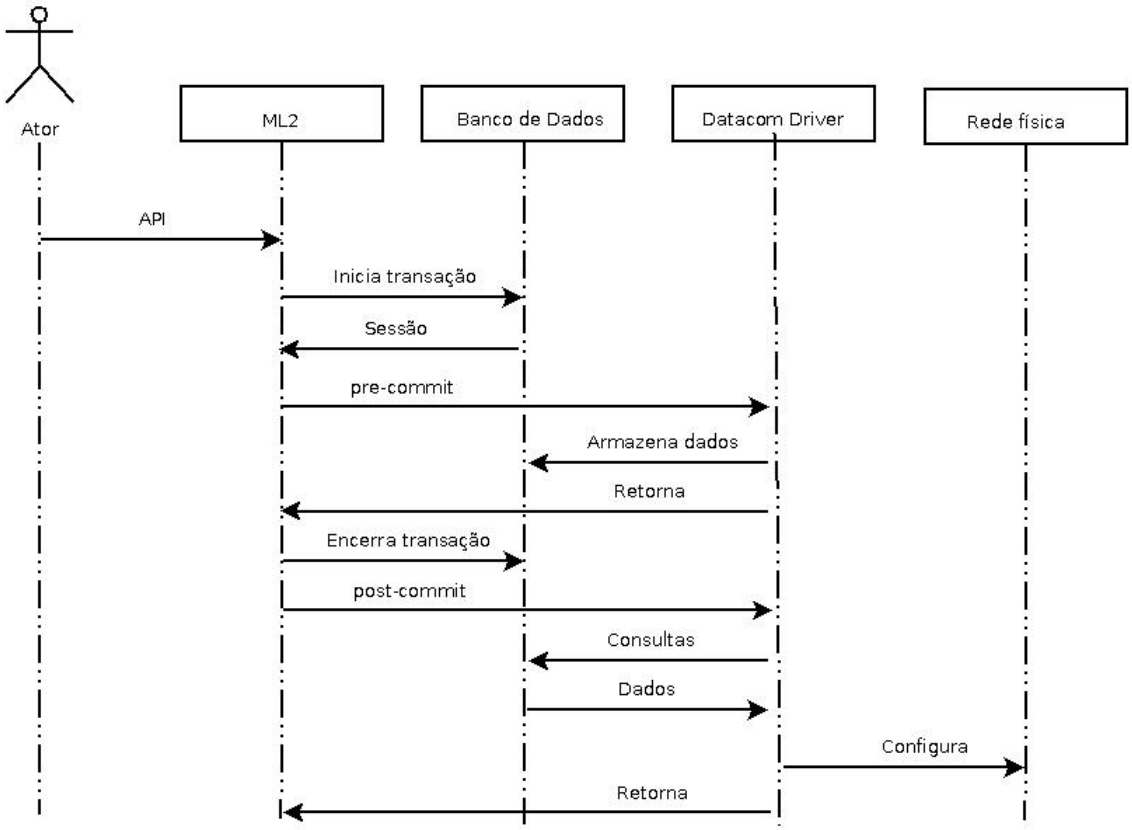

Figura 4.5: Caso de uso genérico da API do driver

4.3 Caso de uso de um driver genérico

Para melhor entender o fluxo de uma requisição ao driver abstrai-se um caso de uso genérico para o uso da API do driver Este caso de uso se encontra na Figura 4.5.

O fluxo desta figura ocorre da seguinte maneira. Primeiro o ator invoca alguma API do *Neutron* relacionada a algum driver. Para o *plugin* ML2 atender esta requisição, primeiramente este deve atualizar o banco de dados. Para tal, este inicia uma transação com o banco de dados, iniciando uma sessão. O ML2 então inicia o processo de *pre-commit* do driver, enviando a sessão retornada pelo banco de dados. O *driver*, então, realiza a manipulação necessária dos dados e retorna ao ML2, o qual encerra a transação com o banco de dados. Se o processo de *pre-commit* ocorrer com sucesso (indício de que não houve inconsistência no banco de dados), o ML2 inicia o processo de *post-commit* do *driver*. Este realizará consultas no banco de dados e usará tais dados para configurar a rede física. Finalmente, o processo retorna ao ML2 finalizando o caso de uso.

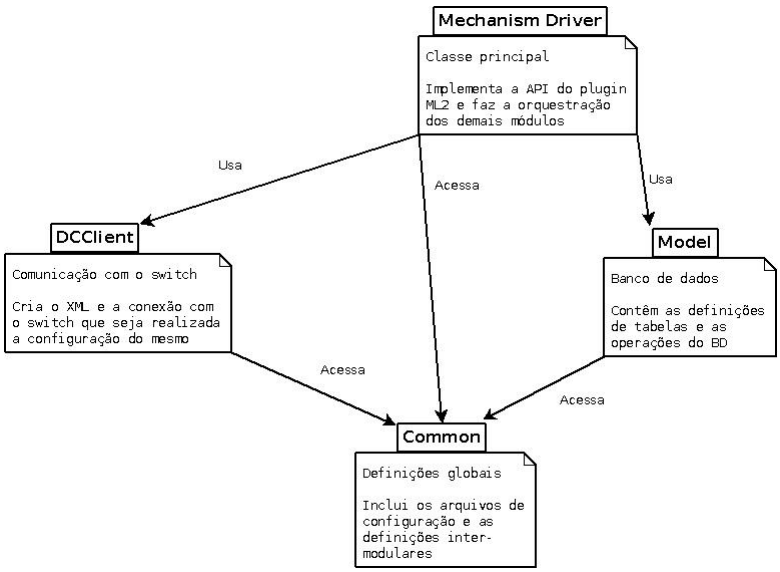

Figura 4.6: Arquitetura genérica do driver

4.4 Arquitetura proposta do driver

A arquitetura proposta, desta vez para o driver, está esquematizada na figura 4.6, possuindo os seguintes componentes:

- **MechanismDriver**: Classe principal do *driver*; implementa a API com o *plugin* ML2 e se comunica com os outros módulos.
- **DCclient**: Biblioteca de comunicação com o comutador Datacom. Contem métodos para configuração do comutador Datacom, como criar uma VLAN e associá-la a portas, e realiza o acesso ao comutador utilizando a configuração determinada pelo usuário. Detalhada na seção 5.2.2.
- **Model**: Tipos de dados utilizados e definição do banco de dados. Mais detalhes são dados na seção 5.1.
- **Common**: Trata-se dos arquivos que contêm configurações e definições usadas em múltiplos módulos, como a versão e definições.

Capítulo 5

IMPLEMENTAÇÃO DA SOLUÇÃO

Este capítulo introduzirá ao leitor a implementação da solução proposta neste trabalho. Assim como o próprio projeto *OpenStack* essa implementação utilizará a linguagem *Python*, em sua versão 2.x. A implementação segue as esquemáticas apresentadas no Capítulo 4 e utiliza diversas bibliotecas para aumentar a robustez do código e garantir sua funcionalidade e modularidade. Dentre as mais importantes bibliotecas utilizadas estão o *SQLAlchemy*, *TestTools*, *Mock*, entre outros.

O capítulo está estruturado da segundo os 3 elementos principais no *driver*, o banco de dados, o módulo que interage com os comutadores e o módulo que interage com o *OpenStack*. De modo a facilitar a leitura estes são apresentados do menos complexo para o mais complexo, começando portanto com o banco de dados e como este está estruturado, seguindo a comunicação com os comutadores e por fim a comunicação com o *OpenStack*. Ao fim do projeto houve a necessidade de criar um arquivo de configuração da rede, este descrito em capitulo próprio devido a complexidade do mesmo.

5.1 Banco de Dados

Para a implementação do banco de dados, foi utilizada a biblioteca *SQLAlchemy*, a qual é utilizada no projeto *OpenStack*. Como SGBD (Sistema de Gerenciamento de Banco de Dados) foi adotado o *mysql*, derivado do plugin ML2 (que, por sua vez, deriva do *OpenStack*). A localização do banco de dados na estrutura do código será dentro do módulo *models*, encontrado na arquitetura proposta na seção 4.6.

Em um *Mechanism Driver* cada uma das operações CRUD é dividida entre *precommit* e *postcommit*. O *precommit* é onde os dados são analizados e logo em seguida guardados no

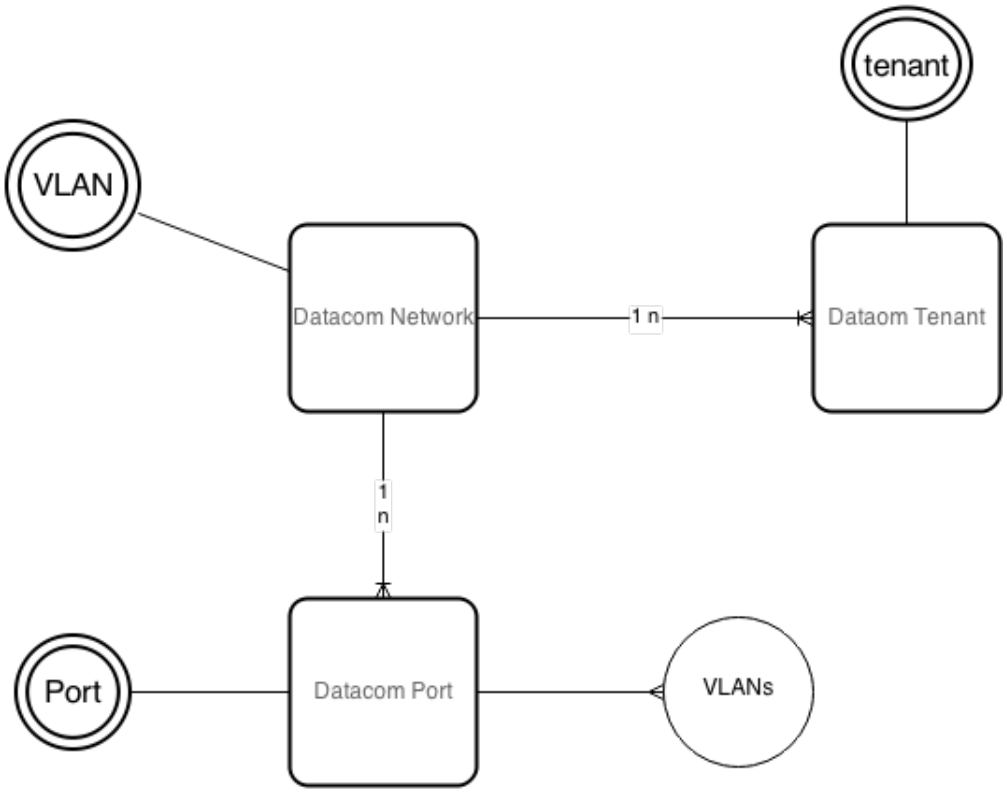

Figura 5.1: Diagrama do banco de dados

banco de dados, já o *postcommit* pode apenas realizar consultas, de qualquer maneira é crucial que a integridade e persistência dos dados seja garantida. O *OpenStack* espera que os *switches* armazenem o seu estado persistentemente e, portanto, o uso de um banco de dados interno torna-se necessário. O módulo *Neutron* não mantém controle das conexões entre *host* e *switch*, bem como a quais portas as *vlans* estarão associadas. Na próxima seção será discutida arquitetura do banco de dados que será utilizado.

O banco de dados projetado para o driver está representado na figura 5.1. Nele estão presentes: a entidade *Datacom Network*, na qual serão guardadas quais redes existem e quais as respectivas VLANs; uma entidade *Datacom Tenant*, que guarda os *tenants* (usuários) e os conecta com as redes; e uma entidade *Datacom port* que guarda os dados das portas físicas dos comutadores e os relaciona também com a rede, conexão da qual pode ser calculado o atributo derivado e multi-valorado *VLANs*.

Além do *schema* do banco de dados, também são criados métodos de acesso para realizar as operações necessárias. Todas as operações são criadas como sub transações, o que garante a consistência do banco de dados.

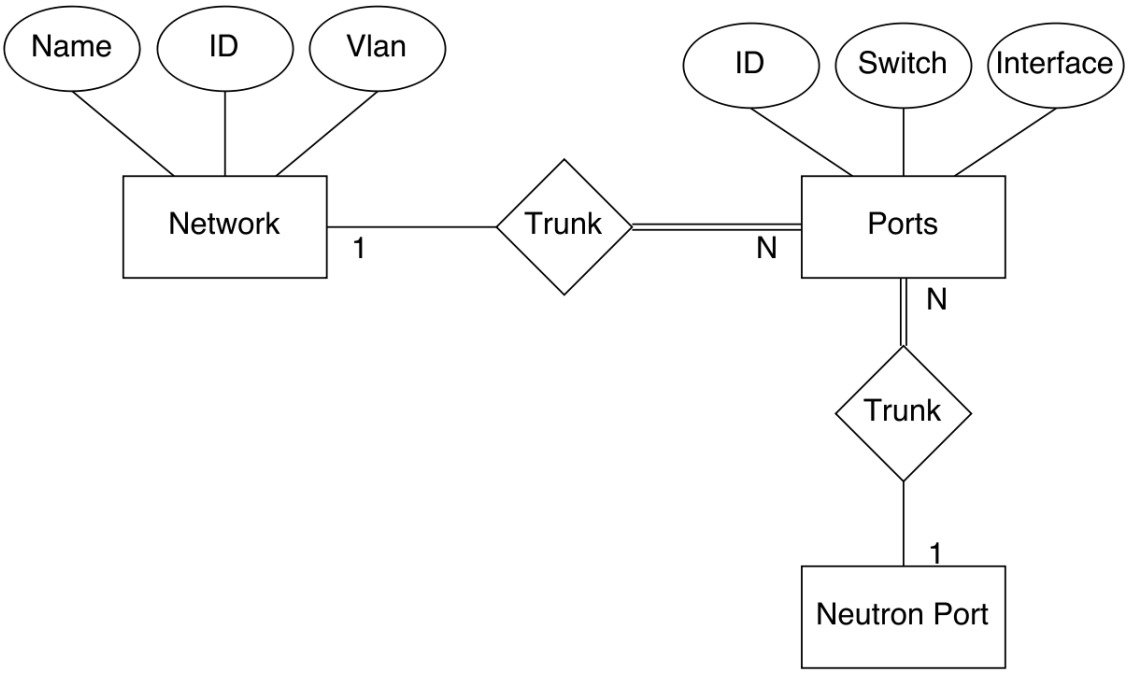

Figura 5.2: Arquitetura do banco de dados

5.1.1 SQLAlchemy

A biblioteca *SQLAlchemy* (SQLALCHEMY..., 2015) é uma biblioteca de ORM (*Object Relational Mapping*) em python. Isto significa que o modelo entidade relacionamento é implementado em objetos, em vez de em SQL. Os objetos cuidam desde da conexão com o banco, até a criação/atualização das classes. No entanto a biblioteca não é apenas ORM, e sim um meio termo entre SQL e ORM, a fim de diminuir significativamente a perda de desempenho devido a abstração. Para se criar e utilizar um banco de dados utilizando esta biblioteca são necessários três passos: a conexão com o SGBD, no qual a conexão com o banco é criada, a criação das classes, na qual as tabelas são mapeadas para objetos, e a criação da sessão, na qual as inserções, deleções e atualizações de dados são realizadas. O primeiro passo é realizado pelo *OpenStack*, de modo que não é necessário ser feito no *driver*, a não ser que o mesmo utilize um SGBD diferente. Para o segundo, a definição das tabelas é feita no módulo *models*, e são carregadas pelo *OpenStack*. O terceiro passo é feito pelo *OpenStack*, mas deve ser realizado também dentro do *driver*, a cada chamada.

5.1.2 Arquitetura

A Figura 5.2 ilustra o modelo entidade relacionamento idealizado para o banco de dados. A primeira tabela, *Network*, representa uma rede do *OpenStack* e possui três campos: o nome da rede, o número da *vlan* (*vid* que representa a rede dentro do comutador) e um identificador único.

Tabela 5.1: Network

nome	tipo	restrição
id	integer	PK
nome	string	
vlan	integer	

Tabela 5.2: Ports

nome	tipo	restrição
id	integer	PK
network_id	integer	FK
ports_id	integer	FK
switch	string	
interface	integer	

A segunda tabela é a *ports*, a qual representa uma interface de um determinado *switch*. Para tanto, ela possui o identificador do *switch* (ou seja, seu *IP*), o número da porta e um identificador único.

A relação dessas duas tabelas é chamada de *trunk*, que representa a ligação de uma *Network* em determinada *port*, de modo que uma rede pode estar ligada a várias portas, caracterizando um relacionamento *many-to-one*. Uma vez que as tabelas estejam dessa maneira é possível, por exemplo, realizar uma consulta no dicionario de comutadores do *driver*, procurando pelo conjunto de porta e comutador, recebendo assim o *host* associado a mesma.

A terceira tabela não consta no banco de dados do *driver*, mas consta no banco de dados do *Neutron*. A função dessa tabela é recuperar quais "portas Datacom", estão relacionadas com uma "porta *Neutron*".

Construído desta maneira, o banco de dados consegue refletir a configuração de cada *switch*, sendo assim possível restaurá-la a partir do banco de dados em caso de perdas. Portanto, é possível regerar a configuração dos *switches* a partir do arquivo *XML* (gerado durante a execução) ou do banco de dados (gerado na inicialização). A seguir é apresentado o processo de implementação do banco de dados.

5.1.3 Implementação

Para a criação e manipulação do banco de dados é utilizado o *SQLAlchemy*. Esse já foi explicado na seção 5.1.1, em resumo, é utilizado para fazer a transformação de objetos em tabelas e vice versa.

O *Neutron* possui um modelo próprio de banco de dados que deve ser utilizado, que prevê o uso de um atributo *ID*, que será a chave primária, e toda tabela deve possuir esse atributo. No caso a tabela *Network* já possui o atributo *vlan*. Esse atributo é único e não pode ser nulo e, portanto, poderia ser utilizado como chave primária. Porém, todas as tabelas devem utilizar o *ID*,

de acordo com o padrão do *Neutron*.

De acordo com a restrição acima, o modelo entidade relacionamento foi traduzido nas Tabelas 5.1 e 5.2. A tabela *Network* é simples, enquanto a tabela *Ports* possui duas chaves estrangeiras: a primeira é o atributo *network_id* que representa sua ligação com a tabela *Network* e a segunda é o atributo *ports_id* que representa sua ligação com a tabela de portas do *Neutron*.

A definição das tabelas será mostrada mais a frente na seção 5.1.5, e o código completo pode ser encontrado no arquivo: *neutron/neutron/plugins/ml2/drivers/datacom/db/models.py*

Segue abaixo a explicação de como utilizar o banco e dados e de como foi feita a integração do banco de dados do *driver* com o banco de dados do *Neutron*.

5.1.4 Definição do banco de dados e criação das tabelas

O banco de dados no *Neutron* é versionado, existindo um arquivo para cada versão, o qual tem a criação e a remoção de tabelas necessárias para a migração a partir da versão anterior. Cada *driver* pode, por sua vez, ter também um arquivo de atualização para a criação e remoção das próprias tabelas. Esse arquivo deve ter dois métodos: o *upgrade*, que contêm a criação das tabelas, e o *downgrade*, que contém a destruição das tabelas. Enquanto a criação do arquivo próprio do *driver* pode ser customizada, contanto que colocado no diretório certo, que é `neutron/db`, o arquivo de versão deve ser criado por meio do módulo do *Neutron* que é responsável pelo banco de dados (*neutron-db-manage*).

Um exemplo do método *upgrade* é o código 5.1 utilizado para a criação da tabela *Network* do *driver Datacom*, que reflete a tabela 5.1. Enquanto que o *upgrade* cria tabelas, como foi comentado, o *downgrade* destrói uma tabela criada, como visto no código 5.2.

```
1  def upgrade():
2      op.create_table(
3          'datacomnetwork',
4          sa.Column('id', sa.String(length=36), nullable=False),
5          sa.Column('vlan', sa.Integer(),
6                    nullable=False, autoincrement=False),
7          sa.Column('name', sa.String(length=30), nullable=True),
8          sa.Column('ports', sa.Integer(),
9                    nullable=True, autoincrement=False),
10         sa.PrimaryKeyConstraint('id'))
```

Listagem 5.1: Método Upgrade

Por fim é importante detalhar como é feita a troca de versão por meio do *neutron-db-manage*. Cada mudança de versão é chamada de migração, e existem dois tipos de migração: o *upgrade* e o *downgrade*. Ambos recebem um parâmetro, que indica para qual versão o banco de dados

```
1 def downgrade():
2     op.drop_table('datacomnetwork')
```

Listagem 5.2: Método Downgrade

a migração deve ir. Para realizar uma mudança de versão, o módulo deve percorrer todas as versões da atual até a nova executando o *downgrade* ou *upgrade*, dependendo de qual o tipo da migração. O comando básico para executar uma migração é:

```
$ neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/
    neutron/plugins/ml2/ml2_conf.ini [upgrade ou downgrade] [ver_id ou HEAD],
```

Onde *ver_id* é o número da versão para a qual se deseja realizar a migração. A opção *HEAD* pode ser utilizada no lugar do *ver_id* para indicar que o método *upgrade* deve ser realizado até a versão mais atual. Para incluir o arquivo do banco de dados na sequência é necessário, primeiramente, criar um arquivo sequencial ao *script*. Para fazê-lo, utiliza-se o parâmetro *revision*, sendo o comando completo:

```
$ neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/
    neutron/plugins/ml2/ml2_conf.ini revision -m "datacom" --autogenerate
```

Ao realizar o comando, o retorno dele será o arquivo gerado, por exemplo:

```
Generating /caminho/para/o/arquivo ... done
```

O nome do arquivo será do formato *«algum_id>_datacom"*. Após ter sido gerado, deve-se alterar os métodos relativos ao *upgrade* e o *downgrade* para ficar igual ao do *Datacom*, presente no diretório que está um nivel acima (..) (`neutron/db/migration/alembic_migrations/`, fora do versions) chamado `datacom_init_ops.py`. Em seguida, efetua-se o processo de mudança do banco de dados, por meio do comando:

```
$ neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/
    neutron/plugins/ml2/ml2_conf.ini upgrade head
```

Ao executar esse comando as tabelas referentes ao *driver* serão criadas. Se houver necessidade de remover as tabelas, deverá ser utilizado o método *downgrade*. A versão anterior da versão criada está no arquivo de versão criado, na variável *"down_revision"*. Para retornar a versão anterior o comando deve ser o seguinte (sendo *down_ver* a versão encontrada na variável *down_revision*):

```
$ neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/
    neutron/plugins/ml2/ml2_conf.ini downgrade down\_ver
```

Tendo explicado como funciona a configuração do banco de dados, a próxima seção explicará como é feita a definição dos objetos, que devem refletir as tabelas.

5.1.5 Definição dos objetos do banco de dados

A definição dos objetos do *driver* são realizadas no diretório `db/models.py` presente dentro do *driver*. Elas devem ser equivalentes às do arquivo de versionamento que cria as tabelas, e tem a função de mapear os objetos para as entidades. A definição da tabela 5.1, portanto é apresentada da maneira definida no trecho de código 5.3.

```
1 class DatacomNetwork(BASEV2, HasId):
2     __tablename__ = "datacomnetwork"
3
4     vlan = Column(Integer)
5     name = Column(String(30))
```

Listagem 5.3: Classe DatacomNetwork

A definição da tabela 5.2, no entanto difere da tabela em um ponto. Devido a maneira que o modelo do banco de dados do *Neutron* é definido, não é possível definir a restrição (*constraint*) com o banco de dados do *Neutron*. Essa ligação, portanto é feita sem uma restrição, o que é uma péssima prática do ponto de vista de banco de dados Porém, é a única maneira no caso e foi notado que os outros *drivers* também optaram por isso. A tabela ports ficou definida da maneira apresentada no trecho de código 5.4:

```
1  class DatacomPort(BASEV2, HasId):
2      __tablename__ = "datacomport"
3
4      switch = Column(String(36))
5      interface = Column(Integer)
6
7      network_id = Column(String(36),\
8                          ForeignKey('datacomnetwork.id'))
9      network = relationship('DatacomNetwork',\
10                            backref=backref('ports'))
```

Listagem 5.4: Classe DatacomPort

Utilizar esses objetos é semelhante a fazer consultas em *SQL*. Consultas *SQL* são compostas de três partes principais: *select*, *from* e *where*. *Select* define o resultado da consulta, o *From* define em qual tabela ou em qual junção de tabelas a consulta será realizada, e *where* define qual a condição da consulta.

Em *SQL* para saber o nome da rede na tabela *DatacomNetwork* em que a *vlan* é igual a 500, executa-se o seguinte comando *SQL*:

```
select name from DatacomNetwork where vlan = 500;
```

Essa consulta, utilizando o *SQLAlchemy*, fica como exemplificado no trecho de código 5.5

```
1 session.query(DatacomNetwork.name).filter_by(vlan = 500).all()
```

Listagem 5.5: Exemplo simples de consulta

Um exemplo mais complexo é uma junção de duas tabelas. Se o desejado é saber qual interface da tabela *DatacomPort* está ligada em qual rede da tabela *DatacomNetwork*, sendo que o *vlan id* é 500, o seguinte comando é executado em *SQL*:

```
select interface from DatacomPort naturaljoin DatacomNetwork where DatacomNetwork.vlan
    = 500
```

Notar que aqui foi utilizado o *naturaljoin* que é mais eficiente do que utilizar um *join* simples. No *SQLAlchemy* existe uma preocupação muito grande com a eficiência, portanto um *join* será sempre executado da maneira mais eficiente. Dessa forma a consulta é feita como demonstrada no trecho de código 5.6:

```
1 base_query = session.query(DatacomPort.interface)
2 table_query = base_query.join(DatacomNetwork)
3 final_query = table_query.filter_by(vlan = 500)
4
5 resultset = final_query.all()
```

Listagem 5.6: Exemplo complexo de consulta

Nesse exemplo a consulta foi dividida de modo a facilitar a leitura. Primeiro é definido qual será o resultado da consulta, depois é definido o *join*, por fim é definida a restrição.

O *SQLAlchemy* lida com a *session.query* de uma forma eficiente: é retornado apenas a consulta *SQL*. Para que a *query SQL* seja executada, é necessário que a *query* gerada chame o método.all() no final dela. Isto é feito para realizar a query de fato apenas quando os resultados forem necessários, para evitar fazer mais de uma query para realizar a mesma operação. Portanto a busca é realizada apenas na linha da atribuição da variável *resultset*.

Tendo falado sobre todas as partes necessárias para o entendimento do *Mechanism Driver*, a próxima seção tratará do *Mechanism Driver*.

5.2 Biblioteca dccclient

Para a comunicação com o switch será desenvolvida uma biblioteca similar ao *ncclient*. O papel da biblioteca é intermediar o switch e o driver, através de um arquivo XML de configuração, esta biblioteca será um dos módulos do driver.

O formato XML será mapeado por meio da biblioteca *ElementTree*, a qual é explicada na seção 5.2.2. Além da criação e manipulação do arquivo XML, a biblioteca deve realizar também a conexão com o comutador. Para tal, utilizar-se-á das configurações feitas pelo cliente para localizar o IP e as portas dos comutadores. A conexão será feita em HTTPS utilizando um *socket*. O comutador deve ser configurado para receber a conexão, ativando o serviço e configurando o IP e a porta a serem recebido.

Nesta seção a arquitetura geral da biblioteca dcclient é apresentada e os componentes são detalhados.

A arquitetura pode ser visualizada na figura 5.3. Os componentes estão representados por retângulos, as bibliotecas e recursos utilizados estão representados por círculos e recursos externos estão representados por nuvens. As setas ligam um módulo a um submódulo e os traços representam comunicação entre dois componentes. Os detalhes de cada componente presente na arquitetura são descritos nesta seção, bem como as bibliotecas utilizadas. A interação de cada módulo com quaisquer bibliotecas relacionadas também serão descritas na seção específica do módulo.

5.2.1 dcclient

O componente orquestra os demais módulos e garante o funcionamento geral. Ao ser instanciado deve criar os demais módulos e iniciá-los corretamente. Ao receber uma chamada, trata da operação requisitada, criando as sub-operações necessárias, levanta, também, as exceções corretas, caso elas aconteçam.

Este é portanto o módulo principal da biblioteca que realiza, por exemplo, a tradução de uma rede para uma VLAN e atualiza o comutador com as configurações novas. O *Mechanism Driver* deve fazer o acesso ao banco de dados para passar todas as informações relevantes para o *dcclient*, o qual é responsável por montar o arquivo *XML* e enviá-lo para o comutador. Devido ao comportamento do comutador o *dcclient* deve manter o arquivo *XML* e apenas atualizá-lo, reenviando desta maneira as configurações anteriores caso haja alteração.

Note que este módulo não deverá utilizar diretamente o banco de dados, sendo que todas as

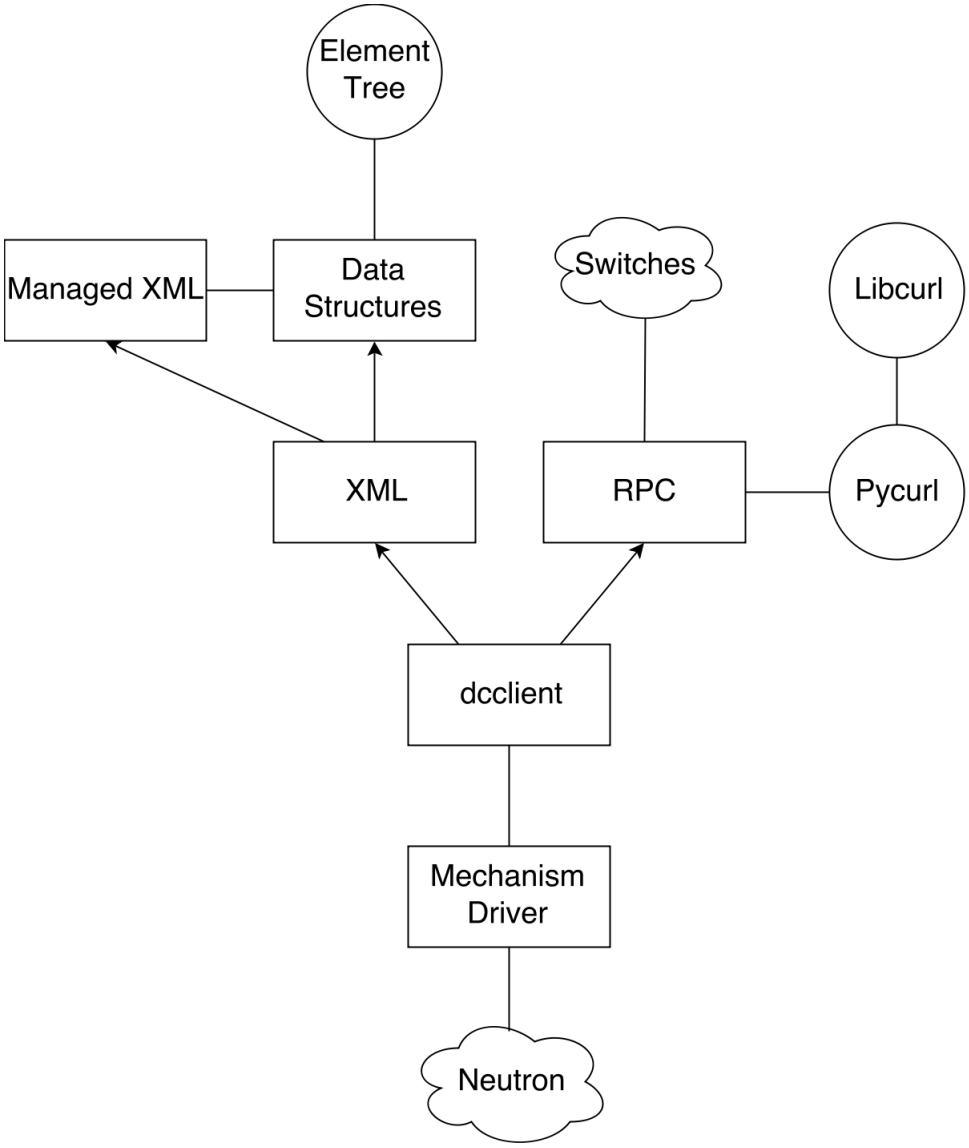

Figura 5.3: Arquitetura da biblioteca dcclient

transações com o banco de dados devem ser realizadas fora dele e apenas as mudanças devem ser transmitidas para o banco de dados.

O método *update port* será demonstrado como um exemplo das operações realizadas por esse módulo. O código final do método *update_port* se encontra no trecho de código 5.7.

Além do parâmetro *self*, o método *update_port* recebe dois parâmetros adicionais: VLAN, que é o número da VLAN; e *ports*, que é um dicionario contendo os *switches* que devem ser atualizados e as portas que devem ser inseridas nesse *switch*.

Assim como os outros métodos dessa classe, esse método trata da montagem do arquivo *XML* que será enviado ao *switch*. No entanto, como mais de um *switch* pode ter sido modificado, o método iterará no dicionário de *switches* modificados construindo um arquivo *XML* para cada

```
def update_port(self, vlan, ports):
  try:
    for switch in ports:
      xml = self.switches_dic[switch]['xml']
      xml.addPortsToVlan(vlan, ports[switch])
    self._update()
  except:
    LOG.info("Trying to add ports to nonexistant network %d:", vlan)
```

Listagem 5.7: *Update Port* do dcclient

um deles adicionando a porta nas *vlans* necessárias.

O método *self._update* utilizado itera no dicionário de todos os *switches* Datacom, atualizando o arquivo *XML* de cada um desses.

5.2.2 XML Manager

O arquivo *XML Manager* (representado na figura 5.3 pelo componente *XML*) é o módulo responsável por gerar o *XML* que será enviado ao switch, e usa a biblioteca *ElementTree* (THE..., 2016) para essa geração. A raiz dessa arvore é guardada em um elemento chamado *root*, além disso, a biblioteca também proporciona métodos de acesso convenientes, por exemplo acessar um campo de vários elementos (por exemplo todos os IPs dos diferentes comutadores). A alteração de qualquer elemento no objeto é facilmente traduzida para o arquivo XML adicionando o atributo *updated* para o elemento.

Neste módulo, as operações são convertidas para o arquivo XML (como a criação de uma VLAN ou a associação de uma porta para uma VLAN específica), deixando o arquivo XML pronto para ser enviado ao switch. Desta maneira, a API deste módulo deve receber um conjunto de operações e recriá-las no arquivo XML, retornando o mesmo para o módulo principal.

Este módulo possui dois arquivos: *data_structures.py* e *manager.py*. O primeiro possui classes que representam os tipos de dados usados no módulo *dcclient* e os tipos de dados específicos do arquivo *XML* a ser enviado, tais como *Pbits*, *Vlan_global* e *Cfg_data*, e manipula essas classes com o ElementTree; o segundo tem o objetivo de ser uma interface entre o módulo *dcclient* e o arquivo *XML*, e um exemplo desse interfaceamento é o trecho de código 5.8.

Este exemplo cria uma vlan (por meio da estrutura de dados *Vlan_global*) e a adiciona à configuração, especificada na estrutura de dados *Cfg_data*. Todas as estruturas de dados possuem o método get_as_xml_text, que retorna o arquivo XML respectivo àquela parte.

A seguir são abordadas as Estruturas de Dados utilizadas pelo *XML Manager*. Estas são *Pbits*, *Vlan_global* e *Cfg_data* respectivamente.

```
vlan = Vlan_global(42)
ports = Pbits([2, 3, 6])

vlan.ports = ports

c = Cfg_data()
c.vlans = [vlan]
```

Listagem 5.8: Interface com o arquivo XML

A classe *Pbits* representa uma máscara de bits das portas do comutador, e possui apenas o atributo *bits*, que é a máscara de *bits*, e possui os métodos para a criação correta da máscara de *bits*. A sua instanciação necessita de um parâmetro que pode ser um inteiro ou uma lista, referente à porta desejada ou ao conjunto de portas desejadas.

A classe *Vlan_global* representa uma VLan. Essa classe possui três atributos: *vid*, que é um inteiro e é o identificador da VLan; *ports*, que é um objeto da classe *Pbits* definida acima; e *name*, que é uma string e também é usada para identificação de uma VLan. A sua instanciação necessita apenas do *vid* e tem *name* e *ports* como parâmetros opcionais.

A classe *Cfg_data* é a classe que contém as configurações das VLans. A sua instanciação não necessita de parâmetros, mas é possível passar uma lista de vlans a serem inseridas. Para a atualização de um objeto de *Cfg_data*, usa-se a atribuição normal, uma vez que na sua definição de classe o método *self.setter* foi sobrecarregado com o uso de *Python @property*.

As estruturas de dados são suficientes, porém por questões de modularidade elas não devem ser utilizadas diretamente. O correto é utilizar-se uma classe que implementa os métodos possíveis com essas estruturas (no caso deste trabalho, o *managed_xml*). Estes métodos serão implementados conforme forem necessários, já que refletem diretamente as operações possíveis no switch (por hora temos 3 métodos). O trecho de código 5.9 cria uma configuração com uma VLAN 42, incluindo as portas 1, 3 e 7.

```
from dcclient.xml_manager.manager import ManagedXml

xml = ManagedXml()
xml.addVlan(42)
xml.addPortsToVlan(42, [1, 3, 7])

print xml.tostring()
```

Listagem 5.9: XML manager example

```
$ python exemplo_xml_manager.py
<cfg_data><vlan_global id0="42"><vid>42</vid><active>1</active><pbmp_untagged id0="0"
    ><pbits id0="0">69</pbits></pbmp_untagged></vlan_global></cfg_data>
```

1. *findVlan*: dado um vid retorna a vlan correpondente se esta existir.

2. *addVlan*: adiciona uma vlan se esta não existir.

3. *addPortsToVlan*: utiliza para adicionar portas a uma vlan existente.

5.2.3 Modelo RPC

Este é o módulo responsável pela comunicação com o conjunto de comutadores. Recebe o arquivo *XML* que será transmitido, cria a conexão e o transmite. Possui apenas uma classe chamada RPC, com dois métodos. O primeiro é um método interno chamado de `__create_url`, e o segundo se chama `send_xml`. O primeiro método é utilizado para montar a url para qual deve ser enviado o arquivo *XML*, dado o IP do switch. O segundo método é o que de fato recebe e envia o arquivo *XML*. Este utiliza a pycurl, detalhada anteriormente. O envio pode ser feito via http ou https. Esta escolha é feita pelo arquivo de configuração do oslo. O arquivo XML, então, passa por uma compressão, e só então é montado o pacote no formato *multipart*, enviado e então a conexão é encerrada.

5.3 Mechanism Driver

Nas seções anteriores foi explicada a importância do *Type Driver* e do *Mechanism Driver*. Relembrando, o *Mechanism Driver* é responsável por receber a informação passada pelo *Type Driver* e aplicar as funções corretas no comutador.

É necessário e suficiente que o *Mechanism Driver* implemente o *CRUD* para as ações que envolvam portas e redes. Com isso obtêm-se um driver totalmente funcional com a versão atual do *OpenStack*.

Cada método dentro do *Mechanism Driver* consiste de duas partes: o *precommit* e o *postcommit*. Como já foi dito anteriormente, o *precommit* é responsável pela interação com o banco de dados, não é bloqueante, e portanto não deve possuir nenhuma interação fora do *Neutron*. Já o *postcommit* é a parte que de fato executa a tarefa, enviando as ações para os *switches*, ou invocando o módulo que o fará.

No entanto antes de detalhar dos métodos em si, será apresentado o fluxo de criação de uma máquina virtual (utilizando o *OpenStack*). Desta forma fica mais simples compreender o funcionamento do *Mechanism Driver*.

5.3.1 Fluxo da criação de uma *VM*

Quando uma nova porta é criada em uma determinada rede, o primeiro método a ser chamado é o *Create Port*. Esse método é repassado do *ML2* para todos os *drivers*. Em seguida, os *drivers*, os agentes (*openvswitch*, o *linuxbridge* e outros possíveis componentes devem se manifestar para realizarem o *bind* na porta que está sendo criada (*Neutron Port*). Esse processo todo se chama *Port Binding*. Ao terminar esse processo, o *ML2* deve então gerar um evento de *Update Port*, passando as portas *Neutron* junto com o *host* ao qual ela se ligou. O *Update Port* de cada *driver* (inclusive o *Datacom*) deve então verificar se o segmento está de fato vinculado e se o *host* deste segmento está conectado em alguma interface do mecanismo gerenciado por esse *driver*. Se essas condições forem satisfeitas, o *driver* deve configurar o mecanismo gerenciado para que corresponda as mudanças realizadas. Na prática, mais de uma *VM* pode estar conectada na mesma porta *Neutron*, que corresponde a interface do *Switch Datacom*. Quando a última *VM* se desconectar, deve-se realizar a configuração do equipamento para que isso seja refletido.

Como o agente é responsável pela ligação da porta com o *host* (*port binding*), torna-se necessária a utilização de pelo menos um agente em paralelo ao *driver*, tal como o *linuxbridge*. Para começar a apresentação dos métodos do *Mech Datacom*, primeiro será apresentada a inicialização do *driver*.

5.3.2 Inicialização do Driver

O método *initialize* do *Mech Datacom* é chamado na inicialização do *driver*, que ocorre na inicialização do *Neutron*. A inicialização do *Mech Datacom* é importante para se proceder a leitura do banco de dados e do arquio de configuração. Os dados recuperados do arquivo de configuração são lidos pelo *dcclient*, como será detalhado na Seção 5.4. As informações do banco de dados são resgatadas e enviadas aos *switches*, para garantir que esses estejam atualizados, A codificação portanto ficou da maneira demonstrada pelo trecho de código 5.10.

```
1 def initialize(self):
2     self.dcclient.setup()
3     session = db.get_session()
4     for vlan,name in .query(DatacomNetwork.vlan,
5                                 DatacomNetwork.name).all():
6         self.dcclient.create_network(int(vlan), str(name))
```

Listagem 5.10: Método Initialize

Tem-se primeiro a chamada para leitura do arquivo de configuração, uma vez que esse passo foi completado, é possível iterar no dicionário de *switches*. O próximo passo, portanto, é garantir

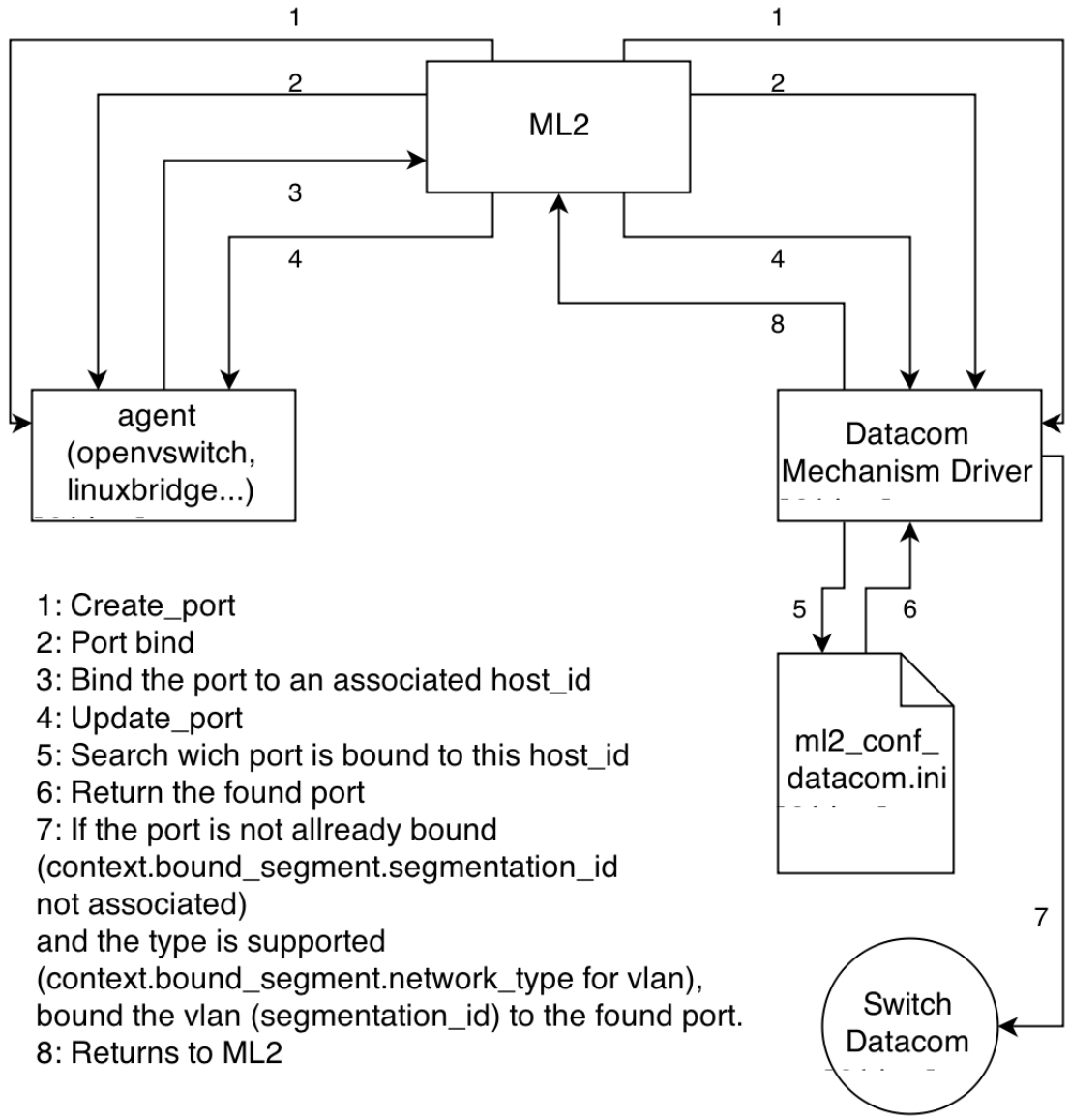

Figura 5.4: Fluxo da criação de porta no *Neutron*

a consistência do banco de dados, efetuando a leitura do mesmo e enviando aos *switches* as regras que esses devem conter.

Agora serão descritos os métodos do *precommit* e *postcommit*, dado que esses são a chave para o funcionamento do *driver*.

5.3.3 precommit

Nesta fase são realizadas as transações do banco de dados. Ao iniciar esta fase o driver deve utilizar a sessão criada pelo ML2 e utilizar os métodos do banco de dados para manipulação de dados. Cada métodos cria sua própria sub-transação. Caso um desses métodos acuse algum

problema, lança-se uma exceção de inconsistência no banco de dados, escalando-a para que a transação sofra rollback e a operação seja cancelada.

Alguns drivers escolhem por não implementar a fase *precommit*, seja por não precisar de banco de dados seja por utilizar o criado pelo ML2, sem a necessidade de adição de nenhum dado. Este projeto inclui o próprio banco de dados e, portanto, deve implementar o *precommit*.

Serão detalhados os métodos importantes do *precommit*. São estes o *Create Network* e o *Update Port*.

O *Create Network precommit* cria a tabela *DatacomNetwork*, como demonstra o trecho de código 5.11

```
1 def create_network_precommit(self, context):
2      = db.get_session()
3     with session.begin(subtransactions=True):
4         dm_network = DatacomNetwork(
5                 vlan = int(context.network_segments[0]['segmentation_id']),
6                 name = context.current['name'])
7         session.add(dm_network)
```

Listagem 5.11: Método *Create Network Precommit*

No código apresentado, a primeira e segunda linha são necessárias sempre que se realiza alguma operação no banco de dados para que a operação é considerada segura. Como se trata do método responsável pela criação da rede, é necessário adicionar a rede ao banco de dados do driver *Datacom*. Portanto, dentro da seção é criada uma rede de nome *dm_network*. Adiciona-se o número da *vlan* que a identifica, retirado do *context* recebido no método, assim como o nome da rede. Finaliza-se a seção com o comando *session.add*, pois essa é uma nova entrada no banco de dados Devido a eficiência do *SQLAlchemy* o método é executado no banco de dados apenas com o ultimo comando.

O *precommit* do método *Update Port* é crucial para o banco de dados. Aqui serão adicionadas as portas necessárias ao banco de dados, bem como suas ligações com a rede. O método é apresentado no trecho de código 5.12:

```
1 def update_port_precommit(self, context):
2     if context.bound_segment is not None and str(context.bound_segment['
          network_type']) == "vlan":
3         ports = self._find_ports(context.host)
4         if ports:
5             self._add_ports_to_db(ports, context)
```

Listagem 5.12: Método *Update Port Precommit*

Nesse código, optou-se criar a porta apenas no método *Update Port* ao invés de criá-la no

Create Port, pois dessa maneira é economizado o acesso ao banco de dados, que pode ser custoso computacionalmente. Como o *Update Port* é chamado logo depois do *Create Port*, durante o processo de *Port Binding* (como foi explicado na seção 5.3.1), não há problemas em criar a porta apenas aqui, já que nessa etapa do processo é sabido a qual *host* a porta estará conectada.

Primeiramente é verificado se houve o *port binding*, significando que a porta está ligada a um *host* (do contrário ela não é interessante). Então é verificado o tipo da rede. Se for uma *vlan*, segue-se com o procedimento. São procuradas as portas relacionadas ao hospedeiro e que foram modificadas nessa operação; se essas existirem, serão adicionadas ao banco de dados por meio do método apresentado no trecho de código 5.13

```
1  def _add_ports_to_db(self, ports, context):
2      vlan = int(context.bound_segment['segmentation_id'])
3      session = db.get_session()
4      with session.begin(subtransactions=True):
5          query = session.query(DatacomNetwork)
6          resultset = query.filter(DatacomNetwork.vlan == vlan)
7          dcnetwork = resultset.first()
8          for ip in ports:
9              for port in ports[ip]:
10                 dcport = DatacomPort(network = dcnetwork,
11                                      switch = ip,
12                                      interface = port)
13                 session.add(dcport)
```

Listagem 5.13: Método _add_ports_to_db

Esse método é responsável por adicionar as devidas portas ao banco de dados. Primeiro é atribuído o valor da *vlan* e assim como nas outras operações deve-se iniciar uma sessão com o banco de dados. Para realizar a ligação com a tabela *Network* deve-se encontrar a tabela correta, e atribuí-la para a variável *dcnetwork*. Como esse método recebe uma lista de portas a serem adicionadas, é necessário iterar nessa lista, e para cada porta é criada uma nova tabela *DatacomPort* com os determinados campos. Somente então é feita a adição da tabela.

5.3.4 postcommit

Nesta fase as alterações devem ser refletidas na rede de fato. Os soft-switches usam uma rede especial chamada local, na qual agentes são utilizados para a configuração. Para os switches físicos uma comunicação deve ser criada com o switch para a realização das operações. Primeiramente o switch deve recuperar as informações necessárias no banco de dados para a realização da operação chamada (utilizando os métodos pré-definidos em models). Em seguida a operação é passada para a biblioteca dcclient para gerar um arquivo XML e transmitir ao switch. O driver deve apenas organizar as informações e enviar para a biblioteca. Como utilizar

as informações e quais operações no switch serão realizados fica a cargo da biblioteca dcclient.

Aqui serão detalhados os métodos que fazem as chamadas do *dcclient*, essas irão executar as devidas ações no *switch*.

Devido a estrutura criada para realizar a interface com o *switch*, o método "*Create Network*"é mais simples, como demonstra o trecho de código 5.14.

```
1 def create_network_postcommit(self, context):
2     vlan = int(context.network_segments[0]['segmentation_id'])
3     self.dcclient.create_network(vlan,
4                                  name=str(context.current['name']))
```

Listagem 5.14: Método *Create Network Postcommit*

Aqui é chamado o método *create_network* da classe *dcclient*, passando para o método a *vlan* e o nome da rede. O método cuidará de transferir o comando para o *switch* correspondente.

Devido ao funcionamento do Neutron, como explicado na seção 5.3.1, o ML2 deve gerar uma chamada ao *update port* logo após o processo de *port binding*. O objetivo dessa chamada é verificar se uma conexão recentemente criada por um *host* passa por um dos *switches* controlados pelo *driver*. Caso passe, o *driver* deve cuidar internamente dessa conexão. A partir daí é recebido o *context* de um *update port* e logo após o context um *port binding*. Se o *host* que participou do *port binding* estiver ligado a um *Switch Datacom*, a porta que esse *host* utiliza no *switch* deve ser adicionada à *network* que o *host* deseja utilizar, ou, neste caso, à *vlan* associada a essa *network*. A codificação desse método portanto é demonstrada no trecho de código 5.15.

```
1  def update_port_postcommit(self, context):
2      session = db.get_session()
3      switches_dic = self.dcclient.switches_dic
4      vlan = int(context.network.network_segments[0]\
5                 ['segmentation_id'])
6      query = session.query(DatacomPort.switch,
7                            DatacomPort.interface)
8      r_set = query.filter_by(neutron_port_id =\
9                              context.current['id']).all()
10     update_interfaces = {}
11
12     for switch, interface in r_set:
13         switch = str(switch)
14         interface = int(interface)
15         if switch not in update_interfaces:
16             update_interfaces[switch] = []
17         update_interfaces[switch].append(interface)
18
19     self.dcclient.update_port(vlan, update_interfaces)
```

Listagem 5.15: *update port postcommit*

Na codificação, assim como anteriormente, é sempre necessário iniciar uma seção com o

banco de dados antes de realizar uma consulta. Então o dicionário de *switches* é atribuído para uma variável local, com o intuito de facilitar seu uso. A vlan será necessária mais a frente e, portanto, essa é resgatada do *context* atual. Logo após ocorre uma busca no banco de dados cujo equivalente em *SQL* seria:

```
1 select switch,interface from DatacomPort where DatacomPort.neutron_port_id =
      context.current['id']
```

Essa consulta retorna os campos *switch* e interface das tabelas *DatacomPort* que possuem o campo *neutron_port_id* igual ao *context.current['id']* especificado. Assim como explicitado na seção 5.1, a variável *neutron_port_id* se refere a ligação da tabela *DatacomPort* com a tabela de portas do *Neutron*. Já o *context.current['id']* se refere ao identificador da porta passada no *context* atual. Portanto com essa consulta pretende-se recuperar o *switch* e a interface de uma determinada porta *Datacom*, se essa foi uma das portas afetadas pelo **port binding**.

A seguir, para cada porta afetada, é montado um dicionário possuindo o *switch* dessa porta e as interfaces a serem modificadas. Por ultimo é chamado o método *update_port* do *dcclient* passando a *vlan* e esse dicionário. Esse método fará as modificações necessárias como citado na seção 5.2.

5.4 Arquivo de configuração do *driver* Datacom

Na versão *IceHouse* do *OpenStack*, que foi a versão utlizada neste projeto, o módulo *Neutron* não possui um sub-módulo para descoberta de rede. Por exemplo, em comparação, o equipamento Cisco possui um desses. No entanto é necessário saber quais portas de cada *switch* estão ligadas na topologia. Devido a esta limitação, os desenvolvedores de *plugins* e *drivers* para o *Neutron* devem escolher entre duas opções. A primeira opção é construir um módulo próprio para descoberta de topologia. Essa opção pouparia trabalho do usuário, porque a descoberta seria feita automaticamente e não haveria necessidade da intervenção do usuário. A outra opção é criar um arquivo de configuração e pedir para o usuário o preencher, demandando menos tempo de desenvolvimento e mais trabalho do usuário.

Segundo o *IRC* do *OpenStack*, existem planos para um módulo de descoberta de rede no release "Kilo"do *OpenStack*. Com esta informação, foi decidido juntamente com a Datacom pela utilização de um arquivo de configuração, e então realizar a atualização quando for lançado o módulo de descoberta de rede.

No caso do *ML2* todos os arquivos de configuração dos drivers ficam no diretório:

Tabela 5.3: Variáveis globais

Formato	Explicação
dm_username=**usuário**	Nome de usuário padrão para login nos *switches*
dm_password=**senha**	Senha padrão para login nos *switches*
dm_method=**http ou https**	Método padrão para login nos *switches*. Pode ser http ou https

```
neutron/etc/neutron/plugins/ml2/
```

Portanto, o arquivo de configuração do *driver* também estará nesse diretório, que será chamado *ml2_conf_datacom.ini*. A seguir temos a explicação da estrutura do arquivo de configuração.

5.4.1 Estrutura do Arquivo de Configuração

Tabela 5.4: Variáveis específicas por *switch*

Formato	Explicação
[***IP* do *switch***]	*IP* do *switch* entre colchetes. As configurações seguintes se referem a esse *switch* até que outro seja encontrado
nome do *host*=porta	Para cada *host* conectado no *switch* é necessário especificar o nome do *host* e a porta a qual está conectado no *switch*
dm_username=**usuário**	Nome de usuário para login no *switch*. Se não for definido a variável global será utilizada
dm_password=**senha**	Senha para login no *switch*. Se não for definido a varável global será utilizada
dm_method=**http ou https**	Método para login no *switch*. Pode ser http ou https. Se não for definido a varável global será utilizada
dm_port=**porta de comunicação**	(opcional) porta utilizada para fazer *login* no *switch*. Por padrão são utilizadas as portas 8080 para http e 443 para https
dm_nports=**número de portas no *switch***	Número de portas (interfaces) que o *switch* possui

O arquivo de configuração possui variáveis globais e variáveis especificas do *switch*. As variáveis globais estão descritas na Tabela 5.3, enquanto as variáveis do *switch* estão descritas na Tabela 5.4. Um exemplo de arquivo de configuração pode ser visto no trecho de código 5.16

As variáveis presentes antes das definições dos *switches* são consideradas globais e, portanto, sempre que um *switch* não definir um *dm_username, admin1* será utilizado. Quando um *IP* de um *switch* é definido entre colchetes os atributos abaixo serão referentes a ele, até que seja definido outro *switch*.

No primeiro caso, todas as variáveis são utilizadas para demonstrar suas funcionalidades. Nota-se que podem ser ligados tantos *hosts* quanto for possível no *switch*, mas para serem reconhecidos os *hosts* devem ser identificados corretamente no arquivo de configuração. Também nesse casso, como a variável `dm_ports` foi utilizada, a comunicação *http* irá ocorrer na porta

```
1  dm_username=admin1
2  dm_password=admin1
3  dm_method=https
4
5  [192.168.1.109]
6  compute1=1/1
7  compute2=1/2
8  dm_username=admin2
9  dm_password=admin2
10 dm_method=http
11 dm_port=2443
12 dm_nports=20
13
14 [192.168.1.110]
15 compute1=1/1
16 dm_nports=20
```

Listagem 5.16: Exemplo do arquivo de configuração

2443 e não na 8080.

O segundo caso consiste em uma demonstração de configuração mais simples e usual, utilizando o máximo possível de atributos globais. Na próxima seção será explicitado como ocorre a leitura do arquivo de configuração.

Vale ressaltar que em versões anteriores do arquivo de configuração era admitido apenas um *Switch Datacom*, o arquivo foi adaptado para a utilização de mais *switches* devido à necessidade da adaptação para este *deliverable*. A versão atual do arquivo de configuração, assim como exemplificado acima, possui suporte a múltiplos *switches*.

5.4.2 Codificação do *parser*

A leitura do arquivo de configuração é realizada no módulo *dcclient.py*. Primeiramente é efetuado a análise do arquivo de configuração pelo método *oslo.config.MultiConfigParser()*, recuperando as configurações de cada *switch* no arquivo e montando um dicionário com as informações de cada um, atentando para configurações globais. A montagem do dicionário pode ser vista no trecho de código 5.17, e esse dicionário é muito importante para a operação de *Update Port*. A chave utilizada no dicionário é o *IP* de cada *switch*. O preenchimento do dicionário pode ser vista no trecho de código 5.18. Para cada *switch* as seguintes informações são guardadas:

- **rpc**: objeto *RPC* instanciado a partir das informações da configuração. Responsável pela transmissão de arquivos *XML* para o *switch*. Cada *switch* tem o seu próprio *RPC* instanciado, com suas configurações (método, porta, etc...)

- **XML**: objeto *XML* que armazena o *setup* dos *switches* (vlans, interfaces e etc...)

```
# create the actual dictionary
for switch in switches:
    sw_dic = config[switch]
    # each field is a list, when it should be a value
    for i in sw_dic:
        sw_dic[i] = sw_dic[i][0]
```

Listagem 5.17: Criando o dicionário de *switches*

```
 # get each global configuration,
 # when not mentioned in the specific
for field in cfg.CONF.ml2_datacom:
    if not field in sw_dic:
        sw_dic[field] = cfg.CONF.ml2_datacom[field]

sw_dic['rpc'] = rpc.RPC(str(sw_dic['dm_username']),
                        str(sw_dic['dm_password']),
                        str(switch),
                        str(sw_dic['dm_method']))

sw_dic['xml'] = ManagedXml()
self.switches_dic[switch] = sw_dic
```

Listagem 5.18: Populando o dicionário de switches

O método principal do *RPC* é o *send_xml*, que recebe um arquivo *XML*, o qual é um arquivo de configuração que será enviado como para o respectivo *switch*. O arquivo *XML* campos para o acréscimo de *vlans* e interfaces para essas *vlans*.

Capítulo 6

AMBIENTE COMPUTACIONAL

Existem diversas maneiras de realizar uma instalação do *Openstack*, assim como existem diversas versões do mesmo. Portanto, a comunidade *OpenStack* foi importante para conhecer mais e decidir qual seria o próximo passo. A comunidade está disponivel pelos seguintes meios: *Ask.OpenStack.com*, pelo canal do IRC *#openstack-neutron @ FreeNode.net*, e pela lista de e-mails *OpenStack-dev*.

Foram obtidos direcionamentos e orientações de desenvolvedores do *Neutron*, por meio do canal do IRC, que auxiliaram na construção e na compreensão do ambiente que segue.

6.1 Ambiente de testes

Para a Implantação foi utilizada uma sala de servidores da UFSCar com o *DevStack* (DEVSTACK..., 2016) instalado.

Para que o *DevStack* encontre o *Datacom Mechanism Driver* e configure o *OpenStack* para utilizá-lo é necessário alterar configurações no arquivo *local.conf* localizado em */opt/stack/devstack*. Mais especificamente, é necessário modificar as seguintes linhas:

```
Q_ML2_PLUGIN_MECHANISM_DRIVERS=datacom
Q_ML2_TENANT_NETWORK_TYPE=vlan.
```

Na linha `Q_ML2_PLUGIN_MECHANISM_DRIVERS` é especificado o(s) *Mechanism Driver(s)* que o *DevStack* carregará na sua inicialização e pode ter os valores da seguinte forma: `Q_ML2_PLUGIN_MECHANISM_DRIVERS=openvswitch,linuxbridge` (neste caso seriam carregados os drivers do *OpenVSwitch* e do *LinuxBridge*). A linha `Q_ML2_TENANT_NETWORK_TYPE` é o *Type Driver* que o *Mechanism Driver* adota. No caso de *Software Switches*, tais como o

OpenVSwitch e o *LinuxBridge*, entre outros, o *Type Driver* é "local", como visto na seguinte linha: `Q_ML2_TENANT_NETWORK_TYPE=local`. Por enquanto o único *Type Driver* suportado por todos os *drivers* de comutadores físicos é o VLAN.

Uma vez que essas variáveis sejam alteradas, então é executado o *script* `stack.sh`, que implanta, configura e executa o OpenStack de forma que, ao finalizar o *script*, o *OpenStack* estará sendo executado com os módulos também especificados no arquivo *local.conf* devido ao número de operações executadas pelo *script*, é possível que este demore para finalizar a execução.

Parte do *stack.sh* configura múltiplas *Screens*, estes são terminais independentes onde são executados os módulos do *DevStack*, um módulo por *Screen*. Isto permite a depuração separada de cada módulo. Uma vez terminado o *script stack.sh*, é necessário executar o *script rejoin-stack.sh* para que se possa visualizar e navegar entre essas *Screens*.

Após inicializados os *scripts* é possível acessar a *Screen* referente ao *Quantum Service*, ou apenas *q-svc*, que é a *Screen* de gerencia um dos serviços do módulo *Neutron*, apesar de ainda manter o nome defasado (*Quantum*). Nessa *Screen* é possível parar o código em execução, o último comando será:

```
user@machine:~/neutron$ cd /opt/stack/neutron && python \
/usr/local/bin/neutron-server
--config-file /etc/neutron/neutron.conf \
--config-file /etc/neutron/plugins/ml2/ml2_conf.ini \
--config-file //opt/stack/neutron/etc/neutron/plugins
/ml2/datacom.ini
```

A partir disto é possível alterar o código do *driver* e após o termino da alteração executar esta linha novamente, de modo a facilitar o *debug* neste módulo sem que isto afete os demais.

6.2 Bare-Metal do Switch Datacom

Para a realização de testes referentes à criação e utilização do arquivo XML, foi gerada uma máquina virtual do tipo *bare-metal* do comutador da Datacom em uma máquina virtual a parte. A imagem da máquina virtual foi enviada pela própria Datacom.

Foi utilizado o KVM para a virtualização dessa máquina. A máquina virtual foi renomeada para *"datacom.img"*. Foi utilizado um endereço MAC como parâmetro e o *script* localizado em */etc/qemu-ifup* que possui comandos para a criação de *bridges* e a criação e habilitação da interface TAP. O comando utilizado para a virtualização é:

```
sudo qemu-system-x86\_64 -enable-kvm \
 -drive file=datacom.img,format=raw -m 2048 -device \
 rtl8139,netdev=net0,mac=CA:FE:CA:FE:CA:FE \newline \
 -netdev tap,id=net0,script=/etc/qemu-ifup
```

6.3 Ambiente de Desenvolvimento

Como máquina de desenvolvimento, foi utilizada uma máquina virtual para possibilitar o desenvolvimento ágil e local, a qual não tem o *DevStack* instalado. O *DevStack* é utilizado apenas na implantação, como descrito na seção 6.1. A máquina virtual de desenvolvimento contem as seguintes ferramentas:

- O código do *OpenStack* com o ambiente de testes;
- O VIM como ferramenta de desenvolvimento, configurado para facilitar o desenvolvimento em *Python*;
- Ferramentas de controle de qualidade de código *Python*;
- repositório GIT, para o versionamento do código.

É mantido um repositório GIT separado para o driver para controle de versão, este será movido para um clone do código oficial do *Neutron*, o qual pode ser encontrado em `https://github.com/openstack/neutron`. A partir daí, o desenvolvimento é realizado e incorporado no subdiretório do git: *neutron/plugins/ml2/drivers/datacom.*

O VIM foi configurado a partir do arquivo de configuração *.vimrc*, que está localizado na pasta *home* do usuário *root*, e possui vários *plugins* bem como pacotes de codificação. Também foram integradas com o VIM as ferramentas de qualidade de código.

As ferramentas de qualidade de código instaladas foram:

- PEP8: Correção de acordo com as normas PEP8, pode ser encontrado em (PEP..., 2016);
- PyLint: Correção de acordo com algumas normas PEP8 e mais algumas diretivas (como o tamanho da linha e o uso de bibliotecas), pode ser encontrado em (PYLINT...,);
- PyFlakes: Correção de erros lógicos do programa, pode ser encontrado em (PYFLAKES..., 2015).

A ferramenta GIT também foi configurada para manter estes padrões antes de incluir o código permanentemente, utilizando *git-hooks*.

O desenvolvimento foi realizado em uma máquina virtual Fedora 21. A instalação do *OpenStack* foi realizada por meio do *Packstack*, com configuração *all-in-one*. *Packstack* é um programa que realiza a instalação do *OpenStack* em uma ou mais máquinas, sejam elas virtuais ou não. Para a realização de tal instalação, são necessários vários parâmetros de instalação.

Existem três papéis a serem definidos em uma instalação do *OpenStack*: *controller*, *network* e *node*. O *controller* é o papel principal, que gerencia os demais. Alguns componentes deste papel são o *Horizon*, o *Heat* e o *Ceilometer*, esse último que foi removido da instalação do ambiente. O componente *network* é o papel que configura a rede, e que abriga a instalação do *Neutron*, tornando-se assim o componente mais importante para este trabalho. O *node* é um *host*, cujo trabalho é hospedar máquinas virtuais. Alguns módulos executados pelo node são: *Nova* (*compute*), *Neutron* e *Glance*. É importante notar que mais de um papel pode utilizar o mesmo componente, por exemplo, todos devem instalar o *Neutron* para participar da rede e o serviço *Keystone* para a comunicação. A seguir será explicado mais sobre o *Packstack*.

6.4 Packstack

O *Packstack* (PACKSTACK...,) é uma ferramenta que utiliza módulos *puppet* para implantar automaticamente, por meio de *SSH*, diversas partes do *OpenStack* em servidores pré-instalados. No momento a ferramenta é testada apenas em ambiente *Fedora*, *RHEL* e derivados.

O sistema operacional que foi utilizado na instalação foi o *Fedora* e o gerenciador de pacotes é, portanto, o *yum*. E, para garantir a funcionalidade, é recomendado realizar primeiramente o *update* dos pacotes e adicionar um novo repositório:

```
$ sudo yum update —y
$ sudo yum install —y https://rdo.fedorapeople.org/rdo—release.rpm
$ sudo yum install packstack
```

Em alguns casos o *PackStack* deveria encontrar e realizar o download de todos os módulos necessários, porém isso nem sempre ocorre. Os *logs* do *PackStack* são bem detalhados, portanto não é difícil encontrar qual módulo está faltando. No ambiente foi necessário instalar o módulo *serialization* da seguinte maneira:

```
$ pip install oslo.serialization
```

O *PackStack* simplifica a instalação em apenas um servidor ao usar o parâmetro *–allinone* como demonstrado abaixo:

```
$ packstack --allinone
```

Esse comando basicamente instancia todos os módulos do *OpenStack* em uma só máquina, gerando um arquivo de configuração padrão. Uma outra opção é utilizar apenas o comando *PackStack*, que inicia em modo interativo, no qual o arquivo de configuração é montado interativamente. Também é possível utilizar um arquivo de configuração no lugar dos parâmetros. Para criar o arquivo de configuração previamente preenchidos para uma instalação *all-in-one*, o seguinte comando pode ser utilizado:

```
$ packstack --gen-answer-file=datacom.txt
```

Desse modo é escolhido o nome e o local do arquivo de configuração do *Packstack*. É possível personalizá-lo com diversos parâmetros presentes no arquivo, e então utilizá-lo novamente para outro *Packstack* da seguinte maneira, gerando o mesmo ambiente:

```
$ packstack --answer-file=datacom.txt
```

Desta maneira cria-se um ambiente do tipo *all-in-one*. No entanto, o módulo *Ceilometer* do *OpenStack* não permitia que o *PackStack* terminasse de executar. A decisão tomada foi de retirar o módulo, já que o mesmo é utilizado apenas no sistema de cobrança do *OpenStack*. Outra modificação realizada foi tornar as senhas simples para agilizar o acesso ao *Horizon*. O comando final (que também gera o arquivo de configuração) se apresenta da seguinte maneira:

```
$ packstack --allinone --os-ceilometer-install=n --default-password=stack
```

Esse comando, além de inicializar o *OpenStack*, gera um arquivo de configuração que permite inicializar um *OpenStack* de mesmas configurações, e esse arquivo foi renomeado para *datacom.txt*. Dessa maneira é fácil reiniciar os serviços do *OpenStack*, sendo apenas necessário invocar o *Packstack* utilizando o arquivo *datacom.txt* como parâmetro, conforme demonstrado acima. Nota-se que nem sempre é necessário reiniciar o *OpenStack*, e que esse processo é demorado e deve ser minimizado. Normalmente, para a realização de testes e com o *OpenStack* executando, basta reiniciar o *Neutron* que a versão atualizada do *Neutron* será executada. Mais detalhes podem ser encontrados na Seção 6.4.1. Da maneira até então explicada é possível montar um ambiente *OpenStack* completo, mas ainda sem o *driver Datacom*. Na próxima sessão é explicado como realizar essa instalação incluindo o *driver*.

6.4.1 Packstack com o driver Datacom

Primeiramente, o *driver* Datacom possui dois requisitos a mais do que os requeridos pelo *OpenStack* Isso deverá mudar uma vez que o driver seja submetido e aceito, mas por hora é necessário realizar a instalação separada. O primeiro requisito é atualizar a biblioteca *requests*:

```
$ pip install --upgrade requests
```

Então é necessário instalar o pacote *toolbelt* da biblioteca requests:

```
$ pip install requests_toolbelt
```

O próximo passo é adaptar o *Packstack* para que ele inclua o *driver Datacom*. Para tal, é necessário alterar o arquivo `plugins/neutron_350.py` (presente no *python egg* do *Packstack*, no diretório *site-packages do python*), adicionando "datacom"na a lista associada a "OPTION_LIST". Essa linha informa o *Packstack* que existe um *Mechanism Driver* de nome "datacom". Após a instalação do *OpenStack*, por meio do *Packstack* apresentado, é necessário trocar o *Neutron* instalado via *Packstack* pelo *Neutron* presente no GIT do projeto, que está à frente da versão de *release* do *IceHouse*. O caminho da instalação do *Neutron*, no ambiente, é:

/usr/lib/python2.7/site-packages/neutron/

No Git do *driver* Datacom, a raiz é um diretório de nome *"neutron"*, com um subdiretório também de nome *"neutron"* (seguindo o padrão do *OpenStack*). Utilizando o *DevStack* ou o *OpenStack* normalmente é excluído o diretório do Neutron, em seguida realizado o *git clone* da seguinte maneira:

```
$ git clone https://github.com/asgard-lab/neutron
```

No entanto o *Packstack* difere nesse padrão já que o diretório *"neutron"* que deve ser copiado não é o raiz, e sim o sub diretório *"neutron"*. Portanto é necessário copiar esse diretório interno para o caminho acima. Estando no diretório onde foi feito o *git clone*:

```
$ cd neutron
$ cp -r neutron /usr/lib/python2.7/site-packages/neutron/
```

Além disso, é necessário copiar o arquivo localizado no diretório:

```
neutron/etc/neutron/plugins/ml2/ml2\_conf\_datacom.ini
```

Para o diretório:

```
/etc/neutron/plugins/ml2/
```

Esse é o arquivo de configuração usado, explicado na Seção 5.4. Além disso, para que o *PackStack* consiga encontrar o driver *Neutron* também é necessário modificar um de seus arquivos de configuração, o arquivo *entry_points*. No ambiente o arquivo é encontrado da seguinte maneira:

```
$ vim /usr/lib/python2.7/site-packages/neutron*.egg-info/entry_points*
```

Nesse arquivo é necessário identificar a seção:

```
[neutron.ml2.mechanism_drivers]
```

E nela adicionar a linha referente ao driver Datacom:

```
datacom = neutron.plugins.ml2.drivers.datacom.mech_datacom:DatacomDriver
```

O próximo passo é ajustar o banco de dados do *Neutron* para suportar o *driver*. Para isso, cria-se um *script* de banco de dados para a atualização do mesmo por meio do comando:

```
$ neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/
    neutron/plugins/ml2/ml2_conf.ini revision -m "datacom"
```

.

Após realizar o comando o caminho do *script* é apresentado no terminal da seguinte maneira:

```
Generating /caminho/para/o/arquivo ... done
```

O nome do arquivo estará no formato "<*algum_id*>*_datacom*". Após ter sido gerado, deve-se alterar os métodos relativos ao *upgrade* e o *downgrade* presentes nesse *script*, mais informações sobre esses métodos serão apresentadas na Seção 5.4. Os métodos corretos estão no arquivo de migração do banco de dados encontrado na pasta:

neutron/db/migration/alembic_migrations/datacom_init_ops.py

Depois de copiados os métodos do `datacom_init_ops.py` para o *script* deve-se então aplicar o *upgrade* para que sejam criadas as tabelas relativas ao *switch* Datacom. Para realizar o *upgrade* utiliza-se o seguinte comando:

```
$ neutron-db-manage --config-file /etc/neutron/neutron.conf --config-file /etc/
    neutron/plugins/ml2/ml2_conf.ini upgrade head
```

O *upgrade* é necessário, porque o arquivo *models.py* contém apenas o mapeamento de objeto para entidade-relacionamento, não contendo a criação das tabelas em si. O *SQLAlchemy* possui uma função de criação para ser utilizada a partir do *models.py*, porém essa função não é recomendada pois vai contra a organização do próprio *Neutron*, que é de manter o banco de dados versionado. Uma explicação mais detalhada sobre esse assunto pode ser encontrada na seção 5.1.

Após as mudanças relativas ao banco de dados, é necessário alterar o serviço do *neutron-server* para que o arquivo de configuração do *driver* Datacom seja incluído. No caso do *Fedora* a alteração é realizada no arquivo relativo ao serviço do próprio *Neutron*, que é, por padrão, `/usr/lib/systemd/system/neutron-server.service`. Nesse arquivo é encontrada a linha de execução do *neutron-server*, a qual possui alguns arquivos de configuração. O arquivo de configuração do Datacom deve ser adicionado nessa linha, resultando, por exemplo, na seguinte linha:

```
ExecStart=/usr/bin/python /usr/bin/neutron-server --config-file /usr/share/neutron/
    neutron-dist.conf --config-file /etc/neutron/neutron.conf --config-file /
    etc/neutron/plugin.ini --config-file /etc/neutron/plugins/ml2/ml2_conf_datacom.
    ini --log-file /var/log/neutron/server.log
```

Após essa alteração o serviço deve ser atualizado por meio do comando `systemctl daemon-reload`.

6.4.2 Depuração do módulo com *PDB*

Para realizar a depuração do módulo, é necessário incluir o *PDB* no código do driver, no local aonde queira depurar. Se o *neutron-server* for iniciado fora do Packstack, como comentado anteriormente, o comando deve ser o mesmo que o utilizado pelo serviço e o depurador pode ser o *PDB*. Se o *neutron-server* for iniciado pelo serviço, o comando passa a ser `service neutron-server start` ou `systemctl start neutron-server`. Porém, na segunda opção, deve-se utilizar um depurador remoto como o *RDB* para a depuração. Para utilizar um depurador remoto basta inserir o código citado no trecho de código 6.1 (requer o pacote *python-celery*). Após a execução do serviço basta acessar o depurador por meio de *telnet* na porta escolhida (por padrão 6899, pode ser escolhida por meio da variável de ambiente *CELERY_RDB_PORT*).

Para a execução correta do *driver*, algum módulo de *agent* deve ter sido iniciado também. O *PDB* é mais simples de utilizar e mais imediato, mas se o local do código for atingido novamente, o *PDB* pode apresentar problemas que inutiliza o terminal. Em comparação, o *RDB* iniciará

```
1 from celery.contrib import rdb
2 rdb.set_trace()
```

Listagem 6.1: Exemplo RDB

normalmente com uma porta acima da definida anteriormente.

Capítulo 7

TESTES E VALIDAÇÃO

Para a validação do driver foi utilizado o ambiente de desenvolvimento introduzido no capitulo 6. Portanto tem-se basicamente duas "entidades"importantes, a máquina virtual onde está a instalação completa do OpenStack utilizando *Packstack* (também referida como *all in one* por ter todos os componentes em uma mesma máquina, e uma segunda com uma réplica de um comutador Datacom, com o mesmo código que um comutador real utiliza, de modo a replicar fielmente o funcionamento do comutador.

Para garantir que o código esteja funcionando corretamente foram desenvolvidos testes unitários. O uso destes é uma convenção nos módulos do *OpenStack*, devido a sua natureza altamente modular e a própria maneira como o desenvolvimento com código aberto funciona. São duas as maneiras de executar testes unitários no OpenStack: através do TOX ou do *script run_tests.sh*. Dentre as duas, o TOX é a convenção mais nova e portanto o seu uso é preferível em relação ao do *run_tests.sh*. A seção traz exemplos de como executar os testes via TOX e via *script*.

Um teste unitário possui duas principais características, garantir que a nível de código todos os possíveis caminhos são cobertos e garantir que o código está funcionando no nível de módulos. Por esse motivo os testes unitários do código do driver Datacom são divididos por módulos, cada classe de teste testa as funcionalidades de sua respectiva classe do código principal, cobrindo os diversos casos que podem ocorrer em cada método.

Os testes unitários em *Python* utilizam em sua maioria os métodos de asserção, esses métodos são mais detalhados no anexo, no entanto dois importantes exemplos são:

- assertEquals(x, y, 'mensagem de erro') Verifica se os valores dos objetos x e y são iguais, do contrário utiliza a mensagem de erro definida. Neste método o operador utilizado para verificar a igualdade é o ==

- assert_called_with(x, y, ...) Verifica se na chamada do método foram passados os atributos x, y, ...

A linguagem python já possui diversos métodos de asserção por padrão, no entanto neste trabalho eles foram expandidos utilizando a biblioteca *Testtools*.

Outra biblioteca que precisa ser citada é a *Mock*. Em um teste unitário todas as unidades devem ser testadas, deste modo quando um método X está sendo testado e este possui uma chamada para o método interno Y, o método Y deverá possuir o seu próprio teste sendo portanto errado fazer uma chamada ao método Y durante o teste.

A biblioteca *Mock* é utilizada nesse caso, ela intercepta o código no momento da chamada de um método e ao invés de chamar o mesmo, dá o retorno esperado deste método, possibilitando assim que o módulo atual seja testado fora da influência de outros módulos.

Nesta seção serão apresentados os testes realizados, estes são divididos de acordo com os atuais módulos do driver. Os testes entre módulos são bem diferentes, no entanto dentro de um mesmo módulo os testes são similares. De modo a evitar repetições será apresentado um trecho de código de cada um, e explicação dos demais testes. A melhor maneira de se compreender a construção dos testes é começar pela estrutura mais interna, que possui os testes mais simples, este é o *Data structures*.

7.1 Testes de estruturas de dados

Os testes de estrutura de dados são divididos nos seguintes métodos *test_pbits; test_vlan_global; test_cfg_data*.

7.1.1 test_pbits

Este método é o responsável por testar o construtor e os métodos auxiliares do *pbits* como demonstra o trecho de código 7.1. Nas linhas 6 e 8 deste trecho de código é garantido que ambos os diferentes construtores estão retornando o mesmo número (9 no caso); As linhas 13 e 17 testam duas maneiras diferentes de adicionar números ao *pbits*; As linhas 20 e 23 testam duas maneiras diferentes de se remover números do *pbits*; por fim na linha 28 temos o teste de como está o formato do *pbits* quando este é passado para o arquivo xml.

```
def test_pbits(self):
    # test constructor consistency
    int_cons = ds.Pbits(9)
    list_cons = ds.Pbits([1, 4])

    self.assertEquals(9, int_cons.bits,
                      'Pbits int constructor setting wrong number')
    self.assertEquals(9, list_cons.bits,
                      'Pbits list contructor setting wrong number')

    # test auxiliary methods
    list_cons.add_bits([2, 3])
    self.assertEquals(15, list_cons.bits,
                      '')

    list_cons = list_cons+[5]
    self.assertIs(31, list_cons.bits)

    list_cons = list_cons-[5]
    self.assertIs(15, list_cons.bits)

    list_cons.remove_bits([1, 4])
    self.assertIs(6, list_cons.bits)

    # test xml
    expected_xml = '<pbits id0="0">6</pbits>'
    actual_xml = list_cons.as_xml_text()
    self.assertEquals(expected_xml, actual_xml)
```

Listagem 7.1: teste do data structures que testa os pbits

7.1.2 test_vlan_global

Esse método testa a estrutura de dados *vlan_global*. Primeiro é testado se a VLAN é instanciada com o valor correto, se é instanciada vazia e ativa por *default*. Em seguida é testado se a VLAN pode ser inicializada como inativa, se pode ser inicializada com portas ou com nome; Então é testado se a VLAN pode se tornar inativa, se o nome pode ser modificado, se o ID pode ser modificado e se as portas podem ser modificadas; Por fim o método testa se o arquivo XML retornado pela VLAN é igual ao esperado.

7.1.3 test_cfg_data

Este método testa a estrutura *cfg_data*, essa estrutura armazena o arquivo XML que será enviado ao switch. Aqui é criada um *cfg_data* e uma vlan, a vlan é então associada ao *cfg_data*, e por fim o método testa se o xml contém a vlan como esperado. A vlan então é removida e outro teste é conduzido para verificar se esta não está mais presente no arquivo XML gerado.

7.2 Testes da classe manager

Os testes da classe manager diferem dos testes das estruturas de dados, pois nessa classe os métodos utilizam as estruturas já criadas Portanto, cada método que manipula essas estruturas é testado. Os testes presentes são: *test_as_xml; test_tostring; test_findVlan; test_addVlan; test_removeVlan; test_addPortsToVlan; test_removePortsFromVlan*.

7.2.1 test_as_xml

Esse método é simples, ele apenas testa o método *as_xml*, que por sua vez tem a função de retornar a classe manager no formato XML.

7.2.2 test_tostring

O método *tostring* é similar ao método *as_xml*, a diferença é que *tostring* retorna o XML em um formato mais legível (utilizado em *logs*) Este teste portanto testa essa *string* retornada.

7.2.3 test_findVlan

Como demonstra o trecho de código 7.2 esse método testa o método *findVlan*. A linha 6 testa se o tipo *None* é retornado quando nenhuma VLAN é encontrada; A linha 12 testa se um objeto do tipo *vlan_global* é retornado quando uma VLAN é encontrada; por fim, na linha 20 é testado se o documento XML da VLAN encontrada está correto (utilizando o método *as_xml* que já foi testado).

7.2.4 test_addVlan

O método *addVlan* adiciona uma VLAN para um objeto do tipo *ManagedXml*. Este método testa tal adição. Primeiro há um teste para garantir que a exceção correta é acionada caso haja uma tentativa de adicionar uma VLAN que já existe. Então, há a checagem do documento XML quando é adicionada uma VLAN vazia é adicionada, seguido do teste com uma VLAN contendo todas as informações.

```
1  def test_findVlan(self):
2      # tests if none type is return when vlan is not found
3      xml = mg.ManagedXml()
4
5      answer = xml.findVlan(61)
6      self.assertEquals(answer, None)
7
8      # tests if Vlan_global object is return when vlan is found
9      xml.addVlan(60, name='vlan_test', ports=[1, 5, 9])
10
11     answer = xml.findVlan(60)
12     self.assertIsInstance(answer, ds.Vlan_global)
13
14     # tests if found vlan xml is correct
15     expected_xml = '<vlan_global id0="60"><vid>60</vid><active>1' + \
16                    '</active><name>vlan_test</name>' + \
17                    '<pbmp_untagged id0="0"><pbits id0="0">273</pbits>' + \
18                    '</pbmp_untagged></vlan_global>'
19
20     self.assertEquals(expected_xml, answer.as_xml_text())
```

Listagem 7.2: teste do manager que testa o método findVlan

7.2.5 test_removeVlan

O método *removeVlan* é similar ao *addVlan*, exceto que remove uma VLAN ao invés de adicionar uma. O teste primeiro tenta remover uma VLAN que não existe e checa se a exceção certa é acionada; Então uma VLAN válida é removida e o documento XML resultante é comparado com o XML esperado.

7.2.6 test_addPortsToVlan

O método addPortsToVlan adiciona uma ou mais portas a uma VLAN. Primeiramente é testado se a exceção correta é acionada caso a vlan não existal. Por fim, verifica se o documento XML segue o formato esperado quando portas são adicionadas.

7.2.7 test_removePortsFromVlan

Muito similar ao método acima, primeiro é testada se a exceção correta é acionada caso a vlan não exista, e a seguir, é verificado se o documento XML é igual ao esperado quando uma porta é removida.

7.3 Testes do módulo RPC

RPC é o modulo que "conversa"com o switch Datacom, este módulo possui apenas um método send_xml que será testado da seguinte maneira.

7.3.1 test_send_xml

O método send_xml é responsável por receber um arquivo cfg, derivar dele o endereço do switch, enviar o arquivo e esperar a confirmação de recebimento pelo switch. Como os testes estão sendo apresentados em ordem de complexidade, este é o primeiro teste onde pode-se ver o uso da biblioteca *Mock*, como demonstra o trecho de código 7.3.

Primeiramente tem-se a inicialização de variáveis e a inicialização da classe RPC. Na linha 8 o módulo RPC recebe uma chamada da biblioteca *mock*, essa irá agir no envio do documento XML, pois como um teste está sendo realizado não teremos nenhum comutador de fato conectado. Logo após é criado um arquivo de configuração simples e na linha 15 o *send_xml* é chamado. Na linha 20 o método *mock_calls* é utilizado para retornar as chamadas feiras com o método que recebeu o *mock*, onde "0"passado significa que estamos nos referindo a primeira chamada (e única nesse caso) o "2"significa que estamos nos referindo as palavras chaves do parâmetro. A linha 27 então testa se o método decodificou corretamente a URL do comutador. O próximo passo é testar se o documento XML está correto, para isso utiliza-se o ultimo parâmetro, que está comprimido, e se extrai o documento XML. A ultima linha (51) testa se o documento XML é o mesmo que o esperado.

7.4 Testes do driver dcclient

No código do driver a classe *dcclient* é chamada pelo *mech_datacom* que realiza a orquestração da comunicação com o switch, fazendo uso portanto das classes manager e RPC. Os métodos de teste serão os seguintes: *test_update; test_create_network; test_delete_network; test_update_port; test_delete_port; test_setup*.

Notar que o *test_setup* está por ultimo de propósito, pois é o método mais complexo até agora. Esses testes em sí são mais complexos, pois fazem o uso do *Mock*, bem como das estruturas de dados.

```
def test_send_xml(self):
    test_url = '1.1.1.1'
    test_auth = ('user', 'pass')
    test_method = ('https')

    test_rpc = rpc.RPC(test_auth[0], test_auth[1], test_url, test_method)

    rpc.get = mock.Mock()
    cfg = ds.Cfg_data()

    vlan = ds.Vlan_global(42, name="vlan_test", ports=ds.Pbits([1, 3, 4]))

    cfg.vlans.append(vlan)

    test_rpc.send_xml(cfg.as_xml_text())

    # mock_calls returns the calls executed to this method, the 0 means we
    # are getting the first (and only) call, and the 2 means we are getting
    # the keyword parameters.
    parameters = requests.post.mock_calls[0][2]

    # retrieve url
    received_url = parameters['url']

    expected_url = 'https://1.1.1.1/System/File/file_config.html'

    self.assertEquals(expected_url, received_url)

    # retrieve data from the parameters.
    data = parameters['data']

    expected_xml = '<cfg_data><vlan_global id0="42"><vid>42</vid>' + \
                   '<active>1</active><name>vlan_test</name>' + \
                   '<pbmp_untagged id0="0"><pbits id0="0">13</pbits>' + \
                   '</pbmp_untagged></vlan_global></cfg_data>'

    # Since the XML has to be the last parameter to be passed, we have to
    # get the last field.
    zippedXML = data.fields[-1][-1][-2]

    # decompress to do the comparison.
    # get the file with the zipped xml as content
    zipFileObject = sio(zippedXML)

    # get the actual file (from which we read)
    with gzip.GzipFile(fileobj=zipFileObject, mode='r') as zipFile:
        received_xml = zipFile.read()

    zipFileObject.close()

    self.assertEquals(expected_xml, received_xml)
```

Listagem 7.3: teste do rpc que testa o método send_xml

7.4.1 test_update

O método *update* é utilizado para atualizar o documento XML de acordo com as informações do dicionário de comutadores e então enviar essas informações para o RPC. Para testar esse método primeiro deve-se utilizar o *Mock* no envio para o comutador *(requests.post = mock.Mock())* similar ao que foi feito anteriormente. O *dcclient* então é inicializado passando para o mesmo um arquivo de configuração com *Mock*. então fazemos a chamada do *setup* e do *update* Logo após temos a verificação de chamadas do *update*. Como o arquivo de configuração com o *mock* possui dois comutadores, o *update* deve ser chamado duas vezes.

7.4.2 test_create_network

Create network é o método que atualiza o documento XML com dados de uma nova VLAN. Para testa-lo, como demonstra o trecho de código 7.4 assim como no método anterior (e em todos os outros da módulo *dcclient*), o *dcclient* é inicializado com um *mocked* arquivo de configuração, linha 5. Então o método *create_network* é chamado na linha 9 e temos duas verificações, uma na linha 10 que verifica se o método foi executado duas vezes (uma para cada switch) e a outra na linha 12 que verifica se o método seguinte, o *addVlan*, foi chamado com os parâmetros corretos.

```
1  @mock.patch('%s.MultiConfigParser' % cfg.__name__)
2  def test_create_network(self, mockedparser):
3      ManagedXml.addVlan = mock.Mock()
4
5      test_manager = base_manager(mockedparser)
6
7      test_manager.setup()
8
9      test_manager.create_network(14, name = 'test_vlan')
10     self.assertEquals(ManagedXml.addVlan.call_count, 2,
11                       'expected create network to be called twice')
12     ManagedXml.addVlan.assert_called_with(14, name='test_vlan')
```

Listagem 7.4: teste do dcclient que testa o método create_network

7.4.3 test_delete_network

Contrário ao *create network*, o método *delete network* deve retirar uma rede do documento XML. Como não é possível retirar uma VLAN do arquivo, esta é apenas marcada como inativa. Para testar o método primeiro ocorre o mesmo *setup* do método anterior, então o delete *network* é chamado. Após sua chamada há duas verificações, uma para saber se o método foi chamado duas vezes, e outra para saber se o método seguinte, *removeVlan*, foi chamado com os parâmetros corretos.

7.4.4 test_update_port

O método *update port* atualiza o documento XML de uma VLAN com um dado dicionário de portas. Para testar esse método primeiro tem-se a usual inicialização seguida da chamada do método. Então ocorre a checagem do número de chamadas, que novamente deve ser dois e, por fim, a verificação se os parâmetros foram passados corretamente para o método *addPortsToVlan*.

7.4.5 test_delete_port

Delete port é o método oposto do *update port*, onde o documento XML da VLAN existente deve ser atualizado removendo o dicionário de portas que for passado. O teste desse método é muito similar aos anteriores. Após a inicialização e a chamada do *update port* há a checagem da quantidade de chamadas (que novamente deve ser duas pois há dois comutadores no arquivo de configuração). Por fim, é verificado se os parâmetros corretos foram passados para a função seguinte *removePortsFromVlan*.

7.4.6 test_setup

Finalizando os testes do módulo *dcclient* temos o teste mais complexo até agora. O *setup* é o primeiro método do *dcclient* que é chamado. Sua função é a de realizar a analise do arquivo de configuração e atualizar o dicionário de comutadores, que por sua vez é utilizado nos outros métodos da classe, pois cada ação deve ser replicada em todos os comutadores da rede.

Para testar esse método, primeiro ocorre uma inicialização com arquivo de configuração passado com valores incorretos e logo após ocorre a verificação de qual erro foi mostrado. Então temos a passagem de um arquivo de configuração com apenas um switch configurado. Para verificar se a leitura ocorreu corretamente inspecionamos o dicionário de comutadores se este possui apenas um switch com os parâmetros corretos (verificação positiva). Logo após ocorre um teste similar ao anterior, porém dessa vez passando dois comutadores no arquivo de configuração e garantindo que ambos estão configurados no dicionario de comutadores. Por fim temos o teste onde ao invés de configurar os comutadores um a um, o arquivo de configurações utiliza a configuração global, para proceder a validação dessa configuração no dicionário de comutadores.

Capítulo 8

CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou uma sucinta análise do cenário dos *softwares* para virtualização, inicialmente apenas para sistemas mas hoje incluindo também a virtualização de toda a rede. Dentre as possíveis tecnologias este trabalho focou em uma das mais conceituadas e utilizadas, o *OpenStack* e sobre o seu módulo de redes, o *Neutron*. Este módulo agrega soluções de diversos grandes fabricantes de comutadores, como Brocade, Cisco e Arista. No entanto, não possui nenhuma solução nacional. Este trabalho portanto tratou da construção da primeira solução nacional para a virtualização de redes no *OpenStack*, através da construção de um *driver* para o seu módulo de redes, o *Neutron*.

Para a implementação foi utilizada a linguagem Python em sua versão 2.x, que é a linguagem utilizada no *OpenStack*. Durante o desenvolvimento foram utilizados diversos diagramas para prova de conceito. Como validações foram utilizados testes unitários (estes são necessários para a submissão do código) e por fim foram necessários conhecimentos de sistemas Linux para estabelecimento do ambiente necessário para testes e desenvolvimento. Todos estes elementos combinados resultaram em uma arquitetura bem estruturada e mais modular, diferente dos outros *drivers* presentes para o *Neutron*.

Através deste trabalho teremos um importante avanço para a visibilidade nacional na virtualização de redes, construindo um *driver* para os comutadores da empresa Datacom, inclusive com o apoio da mesma, ao prover uma versão virtualizada do software de um dos seus comutadores. Este driver será enviado para a validação e inclusão no *OpenStack*, devido a licença aberta desta plataforma a implementação estará aberta para a comunidade, podendo ser usada de referencia para futuras tecnologias ou ser melhorada.

Durante o desenvolvimento da solução apresentada deparamos com algumas dificuldades. A primeira delas veio na mudança de planos. Inicialmente era planejada a construção de um *plugin*.

No entanto, o estudo demonstrou que o *plugin* esta caindo em desuso por, entre outros motivos, não possuir suporte a um ambiente heterogêneo. A segunda das dificuldade é em relação as tecnologias utilizadas tanto na programação quanto no ambiente. Devido a presença de diversas tecnologias por vezes foi difícil de manter o correto funcionamento de todas. No entanto, no final foram cumpridas as expectativas com relação a solução proposta por este trabalho.

Por ser uma primeira versão da implementação esta possui todas as operações básicas necessárias para um *driver*, estabelecendo uma fundação sólida. No entanto existem muitos espaços para melhorias, por exemplo, realizar um estudo sobre o impacto do *driver* e medir o seu *overhead* de modo a melhorar a performance das áreas de maior impacto. Outro trabalho futuro seria a implementação de funções além das básicas, sendo que algumas delas necessitam de melhorias por parte da Datacom. O driver no momento suporta apenas o método de VLAN para a separação de redes, outra melhoria seria possibilitar outros métodos, como por exemplo GRE.

Além destas melhorias, o *OpenStack* está em constante desenvolvimento, de modo que é comum drivers funcionarem apenas em versões específicas. Deste modo é desejável que o *driver* desenvolvido possua um suporte continuado, com o objetivo de estar sempre presente nas versões mais recentes do OpenStack. Por exemplo, no momento da construção deste projeto não havia nenhuma tecnologia nativa para descoberta de redes. Portanto ainda é necessário escrever um arquivo de configuração para que o *driver* reconheça os dispositivos como já existem planos para uma descoberta de redes, seria interessante suportar essa funcionalidade no futuro.

REFERÊNCIAS

APACHE CloudStack: OpenSource Cloud Computing. 2015. Acessado em 2016.01.12. Disponível em: <https://cloudstack.apache.org/>.

BENSON, T. et al. Cloudnaas: A cloud networking platform for enterprise applications. In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. New York, NY, USA: ACM, 2011. (SOCC '11), p. 8:1–8:13. ISBN 978-1-4503-0976-9. Disponível em: <http://doi.acm.org/10.1145/2038916.2038924>.

DEVSTACK - an OpenStack Comunity Production. 2016. Acessado em 2016.01.11. Disponível em: <http://docs.openstack.org/developer/devstack/>.

GE, X. et al. Openanfv: Accelerating network function virtualization with a consolidated framework in openstack. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 353–354. ISBN 978-1-4503-2836-4. Disponível em: <http://doi.acm.org/10.1145/2619239.2631426>.

HOME » OpenStack Open Source Cloud Computing Software. 2016. Acessado em 2016.01.12. Disponível em: <https://www.openstack.org/>.

HPE Helion Eucalyptus | Hewlett Packard Enterprise. 2016. Acessado em 2016.01.12. Disponível em: <http://www8.hp.com/us/en/cloud/helion-eucalyptus-overview.html>.

MOGUL, J. C.; POPA, L. What we talk about when we talk about cloud network performance. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 42, n. 5, p. 44–48, set. 2012. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2378956.2378964>.

NEUTRON/NEUTRON/PLUGINS/ML2/DRIVERS at stable/kilo - openstack/neutron - GitHub. 2015. Acessado em 2016.01.11. Disponível em: <https://github.com/openstack/neutron/tree/stable/kilo/neutron/plugins/ml2/drivers>.

OPENSTACK API Documentation. Acessado em 2016.01.12. Disponível em: <http://developer.openstack.org/api-ref-networking-v2.html>.

OPENSTACK Docs: Networking. Acessado em 2016.01.12. Disponível em: <http://docs.openstack.org/admin-guide-cloud/networking.html>.

PACKSTACK - OpenStack. Acessado em 2016.01.12. Disponível em: <https://wiki.openstack.org/wiki/Packstack>.

PEP 0008 – Style Guide for Python Code. 2016. Acessado em 2016.01.16. Disponível em: <https://www.python.org/dev/peps/pep-0008/>.

PROJECT Savanna. 2013. Acessado em 2016.01.09. Disponível em: <http://insights.dice.com/2013/04/19/project-savanna-brings-together-openstack-and-apache-hadoop/>.

PYFLAKES 1.0.0 python package index. 2015. Acessado em 2016.01.12. Disponível em: <https://pypi.python.org/pypi/pyflakes>.

PYLINT - Code Analysis for Python. Acessado em 2016.01.15. Disponível em: <http://www.pylint.org/>.

SQLALCHEMY - The Database Toolkit for Python. 2015. Acessado em 2016.01.13. Disponível em: <http://www.sqlalchemy.org/>.

THE ElementTree XML API. 2016. Acessado em 2016.01.11. Disponível em: <https://docs.python.org/2/library/xml.etree.elementtree.html>.