UNIVERSIDADE ESTADUAL DO CEARÁ

CENTRO DE CIÊNCIAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO ACADÊMICA

BRUNO LOPES ALCANTARA BATISTA

Detector de Fluxos Conflitantes em Redes OpenFlow Utilizando Técnicas de Inteligência Artificial

Fortaleza-CE

2014

BRUNO LOPES ALCANTARA BATISTA

Detector de Fluxos Conflitantes em Redes OpenFlow Utilizando Técnicas de Inteligência Artificial

Dissertação ou Tese apresentada ao Curso de Mestrado Acadêmico em Ciências da Computação do Programa de Pós-Graduação em Ciências da Computação do Centro de Ciência e Tecnologia da Universidade Estadual do Ceará, como requisito para obtenção do título de Mestre em Ciências da Computação.

Orientação:
Professor Dr. Marcial Porto Fernandez
Professor Dr. Gustavo Augusto Lima de Campos

Fortaleza-CE

2014

A000p Bruno Lopes Alcantara Batista.
Detector de Fluxos Conflitantes em Redes OpenFlow Utilizando Técnicas de Inteligência Artificial / Bruno Lopes Alcantara Batista. – Fortaleza-CE, 2014.
66 p.;il.
Dissertação - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Programa de Pós-Graduação Acadêmica, Fortaleza-CE, 2014.
Orientação:
Professor Dr. Marcial Porto Fernandez
Professor Dr. Gustavo Augusto Lima de Campos.
1. OpenFlow 2. Detecção de Conflito 3. Prolog 4. Agente Lógico 5. HermesNet 6. FloodLight. I. Título.

CDD:000.0

Bruno Lopes Alcantara Batista

Detector de Fluxos Conflitantes em Redes OpenFlow Utilizando Técnicas de Inteligência Artificial

Dissertação ou Tese apresentada ao Curso de Mestrado Acadêmico em Ciências da Computação do Programa de Pós-Graduação em Ciências da Computação do Centro de Ciência e Tecnologia da Universidade Estadual do Ceará, como requisito para obtenção do título de Mestre em Ciências da Computação.

Área de Concentração: Ciências da Computação.

Aprovado em: 25/07/2014:

BANCA EXAMINADORA

Professor Dr. Marcial Porto Fernandez (Orientador)
Universidade Estadual do Ceará - UECE

Prof. Dr. Gustavo Augusto Lima de Campos (Orientador)
Universidade Estadual do Ceará - UECE

Prof. Dr. André Ribeiro Cardoso
Universidade Estadual do Ceará - UECE

Prof. Dr. Cidcley Teixeira de Souza
Instituto Federal de Educação, Ciência e Tecnologia do Ceará - IFCE

Este trabalho é dedicado às crianças adultas que,
quando pequenas, sonharam em se tornar cientistas.

Agradecimentos

À Deus, dedico o meu agradecimento maior, porque têm sido tudo em minha vida.

A minha formação como profissional não poderia ter sido concretizada sem a ajuda de meus amáveis e eternos pais Adilson Alcantara Batista e Mariluce Maria Lopes Alcantara Batista, que, no decorrer da minha vida, proporcionaram-me, além de extenso carinho e amor, os conhecimentos da integridade, da perseverança e de procurar sempre em Deus à força maior para o meu desenvolvimento como ser humano. Por essa razão, gostaria de dedicar e reconhecer à vocês, minha imensa gratidão e sempre amor.

Um agradecimento especial à minha amada Emanuelle Lima Rodrigues, que permaneceu sempre ao meu lado, nos bons e maus momentos. Além de me fazer feliz, ajudou-me, durante todo o percurso de minha vida acadêmica, compreendendo-me e ensinando-me para que eu conquistasse um lugar ao sol.

À todos vocês, meu muito obrigado.

"É melhor lançar-se à luta em busca do triunfo mesmo expondo-se ao insucesso, que formar fila com os pobres de espírito, que nem gozam muito nem sofrem muito; E vivem nessa penumbra cinzenta sem conhecer nem vitória nem derrota."

Franklin Roosevelt

Resumo

O protocolo OpenFlow é uma proposta da iniciativa Clean Slate com o objetivo de definir uma nova arquitetura para Internet onde os dispositivos de de rede sejam simples e que o gerenciamento do plano de controle seja executado em um controlador centralizado. A simplicidade e o caráter centralizados dessa arquitetura a torna confiável e barata. Contudo o protocolo não fornece mecanismos para detecção de conflitos nos fluxos definidos, permitindo que fluxos não alcançáveis possam ser configurados nos elementos de rede, o que pode significar em um comportamento não esperado da rede OpenFlow ou na utilização de recursos dos elementos de rede. Neste trabalho é proposta uma abordagem para detecção de conflitos utilizando-se lógica de primeira ordem para definir possíveis antagonismos e empregar um mecanismo de inferência para detectar fluxos conflitantes antes do controlador OpenFlow instância-los nos swicthes de uma rede OpenFlow.

Palavras-Chave: OpenFlow. Detecção de Conflito. Prolog. Agente Lógico. HermesNet. FloodLight

Abstract

The OpenFlow protocol is a proposal from the Clean Slate initiative aiming to define a new architecture for the Internet where network devices are simple and the management control plane runs on a centralized controller. The simplicity and centralized nature of such architecture makes it reliable and cheap. However, this architecture does not provide mechanisms for conflict detection in the defined flows, allowing unreachable flows can be configured in the network elements and what it can mean in a behavior not expected of OpenFlow network. This work proposes an approach for conflict detection using first-order logic to define possible antagonisms and hire an inference engine to detect conflicting flows before the OpenFlow controller instantiates them in swicthes an OpenFlow network.

Keywords: OpenFLow. Conflict Detection. Prolog. Logical Agent. HermesNet. FloodLight

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

API	Application Programming Interface
BDD	Binary Desicion Diagrams
DiffServ	Differentiated Service
GPL	General Public License
HTCD	Hash-Trie based Conflict Detection
HTTP	Hypertext Transfer Protocol
IntServ	Integrated Service
ID	Identificator (identificador)
IP	Internet Protocol
ISP	Internet Service Provider
IPSEC	IP Security
MAC	Media Access Control
NAT	Network Address Translation
OCD	Ontology based Conflict Detection
PBONM	Policy Based OpenFlow Network Management
PNAD	Pesquisa Nacional por Amostra de Domicílios
SDN	Software Defined Network
ToR	Top of Rack
TCAM	Ternary Content-Addressable Memory
VoIP	Voice over IP
VPN	Virtual Private Network

Lista de símbolos

Γ	Tabela de Fluxos *OpenFlow*.
F	Uma entrada de fluxos em uma dada Tabela de Fluxos *OpenFlow*.
C_i	Restrições dos campos de comparação (matching) de uma entrada fluxo *OpenFlow*.
fv_i	Valor dos campos de comparação (matching).
Υ_a	Conjunto de todos os fluxos que possuem a ação $ainA$.
Φ	Conjunto de todos os conflitos estáticos conhecidos.
ϕ	Definição de um conflito estático.
Ψ	Conjunto do todos os conflitos dinâmicos conhecidos.
ψ	Definição de um conflito dinâmico.
P	Percepção do agente lógico.
S	Estado interno do agente lógico.
A	Conjunto de ações possíveis do agente lógico.

SUMÁRIO

1 Introdução

A Internet evoluiu de uma rede militar e universitária, onde seus usuários eram confiáveis e possuíam conhecimento técnico sobre a mesma, para um fenômeno social que mudou a forma de como o ser humano se comunica, realiza negócios, manipula desastres e realiza operações militares (CLARK et al., 2005) (FELDMANN, 2007).

Segundo dados do Censo Demográfico de 2010, o Brasil possui possui mais de 190,7 milhões de habitantes (IBGE, 2010), enquanto que aproximadamente 77,6 milhões de pessoas entre 10 anos ou mais usam a Internet no Brasil (IBGE, 2012), aproximadamente 41% da população brasileira. Esses números mostram a importância da Internet em nossa sociedade e o seu potencial de crescimento nos próximos anos.

Apesar de sua plena aceitação, as limitações existentes da Internet dificultam a criação de novas aplicações que envolvam segurança, mobilidade e qualidade de serviço. Isso deve-se ao fenômeno conhecido como *"ossificação da internet"* no qual o núcleo da rede está intrinsecamente amarrado aos princípios que nortearam a sua criação, na década de 1970 (BRADEN et al., 2000).

Os princípios que norteavam a Internet eram (FELDMANN, 2007):

- **Ser uma rede dividida em camadas**: Facilitar a implementação ou o reuso de serviços, reduzindo a complexidade promovendo o isolamento de funções, resultando em um forma de estruturar o *design* de protocolos.
- **Utilizar a comutação de pacotes**: Dividir os dados em pacotes, onde cada pacote possuiria um endereço de origem e destino tornando um pacote independente do outro. Isso permite utilizar um mecanismo de roteamento *stateless*, ou seja, não é necessário guardar nenhum estado sobre os pacotes, permitindo a escalabilidade da rede e contribui para a relação custo/benefício.
- **Ser uma rede de redes colaborativas**: As decisões de roteamento são realizadas por uma base de dados de rotas que ficará nos dispositivos responsáveis por esta funcionalidade, onde cada rede interligada irá compartilhar essa tabela informações com as outras redes.
- **Possuir sistemas finais inteligentes**: O núcleo da rede realiza apenas tarefas de comunicação, onde toda a inteligência da rede reside nos sistemas finais, ou seja, se um sistema final deseja enviar um determinado dado de forma confiável o mesmo deve escolher um protocolo mais adequado à essa situação e não à rede.
- **Utilizar o argumento fim-a-fim**: O argumento enfatiza que uma determinada funcionalidade não deve ser fornecida nos níveis inferiores do sistema, a menos que

possa ser completamente implementada nesse nível (entretanto essa regra não é absoluta).

No início, a Internet tinha um objetivo claro e preciso, interligar todos os computadores do mundo e, ao mesmo tempo, prover uma infraestrutura para o surgimento de novas aplicações. Porém, com o passar do tempo, a Internet foi se distanciando de seu principal objetivo, por conta de disputas e interesses diversos (CLARK et al., 2005).

Para alcançar seu objetivo, o projeto inicial da Internet possuía os seguintes objetivos específicos (FELDMANN, 2007) (CLARK, 1988):

- Poder conectar-se a redes existentes;
- Possuir Capacidade de sobrevivência;
- Suportar múltiplos tipos de serviços;
- Permitir o gerenciamento distribuído;
- Possuir um baixo custo operacional;
- Permitir que novos computadores (*hosts*) sejam adicionados com pouco esforço;
- Permitir a contabilização de recursos.

Tais interesses comuns não prevalecem mais na Internet. Atualmente existe uma grande disputa de interesses entre todas as entidades interessadas: os usuários da Internet, provedores de serviços de Internet (*ISPs*), fabricantes de equipamentos de rede, governos e provedores de conteúdo (CLARK et al., 2005).

Outro ponto a ser levado em consideração é a quantidade de remendos que a Internet sofreu ao longo dos anos para se adequar as requisitos atuais de aplicativos e serviços, alguns resultante dos conflitos de interesse sobre a Internet (FELDMANN, 2007) (CLARK et al., 2005).

Extensões extra arquiteturais como o *IP Security* (IPSEC), *Integrated Service* (IntServ) e *Differentiated Service* (DiffServ), possuem um senso arquitetural forte, onde foi levado em consideração pelos seus criadores a anexação desses protocolos à arquitetura da Internet.

Entretanto outras extensões que visavam resolver problemas pontuais de um subconjunto de interessados que formam a Internet (fabricantes, usuários ou provedores de Internet), tais como o *Network Address Translation* (NAT), *mobile IP*, *Virtual Private Networks* VPNs, *Label Switching* e *Web Caches* foram criados para suprir necessidades da Internet que não foram levadas em conta no projeto inicial, mas tais incrementos não

levaram em consideração sua integração com a arquitetura da Internet, o que não consiste, tecnicamente, um melhoramento da arquitetura (BRADEN et al., 2000).

O conjunto de remendos criados para atender demandas pontuais da Internet e o fato de estar fortemente engessada no seu projeto inicial, colocou-a diante de um dilema chamado de *"ossificação"* da Internet, tornando difícil o processo de inovação da rede, atendimento aos requisitos atuais e das futuras aplicações (FELDMANN, 2007) (CLARK et al., 2005).

Diante desse dilema, diversos trabalhos vem sendo apresentados pela comunidade visando resolver esse problema. Alguns trabalhos ainda tentam solucionar os problemas da Internet de uma forma incremental, outros propões que a arquitetura da Internet deve ser repensada do zero (FELDMANN, 2007).

Baseado no princípio de uma arquitetura nova pensada do zero, a Universidade de *Stanford*, sediada nos Estados Unidos da América, criou o programa *Clean Slate* que tem como principal objetivo "reinventar a Internet"superando as limitações arquitetônicas da Internet atual, incorporando novas tecnologias, permitindo uma nova gama de aplicações e serviços e, principalmente, continuar ser uma plataforma para inovações que fomente o crescimento econômico e prosperidade para a sociedade (STANFORD, 2014).

Dentro do programa *Clean Slate*, nasceu o conceito de *Software Defined Network* (SDN), que é um novo conceito de arquitetura de rede dinâmica, gerenciável, adaptativa e de custo baixo. SDN permite que administradores de rede gerenciem serviços de rede através da abstração de funcionalidades de níveis inferiores. Isso é realizado, desacoplando o sistema que toma as decisões sobre para onde os dados serão enviados (**plano de controle**) dos sistemas adjacentes que encaminham os dados para um determinado destino (**plano de dados**) (ONF, 2014).

SDN necessita de um método para o plano de controle possa se comunicar com o plano de dados. O protocolo *OpenFlow*, criado dentro do programa *Clean Slate* é o elemento fundamental para a construção de soluções SDN, provendo a separação física do plano de controle e de dados, além de fornecer uma interface de comunicação entre ambos (MCKEOWN et al., 2008).

O plano de dados ainda permanece nos elementos de rede, tais como *switches* e roteadores, aproveitando a capacidade que esses dispositivos possuem de encaminhar e receber grandes quantidades de dados. Entretanto, o plano de controle é colocado em um servidor chamado de *controlador* que possui um maior poder de processamento, permitindo realizar tarefas antes proibitivas nos elementos de rede.

O *OpenFlow* fornece um protocolo para que o controlador se comunique com os elementos de rede. Esse protocolo é simples e de fácil implementação por parte dos fabricantes de elementos de rede, além de permitir que o funcionamento interno de tais

equipamentos não sejam expostos ao ambiente externo, o que protege a propriedade intelectual de cada fabricante ante seus equipamentos (MCKEOWN et al., 2008).

Em uma rede *OpenFlow* cada elemento de rede possui uma tabela de fluxos de rede (abstração base do protocolo *OpenFlow*). Quando um pacote chega por uma interface de um determinado *switch*, por exemplo, o mesmo realiza um processo de *matching* (comparação) na tabela de fluxos a fim de encontrar alguma ação que defina a decisão a ser tomada pelo switch. Caso a ação a ser tomada não possa ser determinada pela ausência de uma entrada na tabela de fluxos que caracterize o pacote, o mesmo é enviado para o controlador, via protocolo *OpenFlow*, para analisá-lo e determinar a ação a ser tomada.

Após a análise do controlador, o mesmo escreve na tabela de fluxos do *switch* utilizando o protocolo *OpenFlow* um fluxo que atenda os requisitos do pacote analisado, a ação a ser tomada e devolve o pacote para o *switch*. Então o *switch* irá analisar novamente o pacote e, desta vez, encontrará uma ação para esse pacote e os demais que possuam as mesmas características.

Contudo, apesar do protocolo *OpenFlow* fornecer uma maneira inovadora e simples de controlar programaticamente uma rede de computadores, o mesmo não fornece uma maneira de verificar a semântica dos fluxos adicionados pelo controlador nos elementos de rede. A falta dessa funcionalidade pode ocasionar:

- A especificação de fluxos errôneos que irão ocupar a tabela de fluxos dos elementos de rede, ocasionando conflitos entre os fluxos (NATARAJAM; HUANG; WOLF, 2012) (AL-SHAER; AL-HAJ, 2010).
- A utilização da memória TCAM nos elementos de rede (usadas para armazenar fluxos *OpenFlow*) que, apesar da sua velocidade de pesquisa, são caras, ocupam muito espaço físico e consomem muita energia.

Diante desses fatos, reduzir as entradas inválidas na tabela de fluxos de um *switch OpenFlow*, melhora a eficiência do sistema.

1.1 Proposta de Trabalho

O presente trabalho propõe-se a criar um mecanismo de detecção de conflitos, que ficará como um processo residente no controlador *OpenFlow*, capaz de identificar definições de fluxos conflitantes na tabela de fluxos de um *switch OpenFlow*. As definições formais de conflitos serão definidas em uma base de conhecimento, utilizando lógica de primeira ordem para faze-lo. Então uma formalização matemática que represente os conflitos, bem como a tabela de fluxos, além de suas entradas individuais, são necessários para a criação do mecanismo de detecção de conflitos.

A lógica de primeira ordem fornece um poder de expressão suficiente para formalizar toda a matemática. Baseado nessa premissa, necessita-se definir matematicamente as características abstratas que regem uma tabela de fluxos *OpenFlow* e seus respectivos campos do ponto de vista matemático. Assim torna-se possível a utilização da lógica de primeira ordem para definir, de forma pragmática, a definição formal de um conflito no contexto de redes *OpenFlow*.

De posse do modelo matemático abstrato da tabela de fluxos *OpenFlow*, um *software* que realize deduções das sentenças da lógica de primeira ordem de forma automatizada deverá ser implementado. A linguagem de programação *Prolog*, que se enquadra no paradigma de programação em lógica matemática, sendo uma linguagem de uso geral que está intrinsecamente ligada à inteligência artificial e a linguística computacional, além de fornecer o poder de expressão necessário para esta tarefa.

Nesse contexto, será criado um *software* em *Prolog* que realize as deduções das sentenças da lógica de primeira ordem. Na implementação desse software, técnicas de inteligência artificial serão utilizadas, onde resultará, ao seu término, na construção de um agente lógico conforme a definição de Russell and Norving (RUSSELL; NORVIG, 2010).

Para a validação da ferramenta proposta por esse trabalho, o agente lógico será incorporado a um controlador *OpenFlow* e será realizado um experimento com o emulador de redes *OpenFlow Mininet* (LANTZ; HELLER, 2012) onde o controlador *OpenFlow* será direcionado a adicionar diversos fluxos conflitantes nos elementos de rede, podendo avaliar o desempenho da proposta e da taxa de acertos que o agente lógico consegue atingir.

1.2 Trabalhos Relacionados

Recentemente, esse problema vem sendo tratado por alguns pesquisadores, onde algumas metodologias foram propostas, além de ferramentas para dirimir esse problema nas redes *OpenFlow*.

Nos estudos de Mirzaei et. al (MIRZAEI et al., 2013). foi realizada uma modelagem das estruturas internas de um switch *OpenFlow* utilizando a liguagem de especificação *Alloy*. A linguagem *Alloy* é baseada em lógica de primeira ordem e muito utilizada na criação de micro modelos que podem ser automaticamente checados e utilizando o *Alloy Analyzer*.

A tabela de fluxos *OpenFlow* foi modelada as estruturas estáticas e dinâmicas de um switch *OpenFlow* e foi gerado, como resultado, um modelo lógico no qual pesquisadores possam analizar as propriedades de um switch *OpenFlow*. Entretanto o trabalho de Mirzaei et. al ter modelado um switch *OpenFlow* utilizando Alloy, o mesmo não abordou a modelagem de fluxos conflitantes em um rede *OpenFlow*.

No trabalho de Natarajan, Huang e Wolf foram identificados problemas com

conflitos ocultos em uma rede *Openflow* utilizando-se o FlowVisor, mostrando diversas metodologias para identificar e resolver estes conflitos (NATARAJAM; HUANG; WOLF, 2012).

FlowVisor é um controlador *OpenFlow* de propósito especial que atua como um *proxy* transparente entre *switches* e múltiplos controladores *OpenFlow* (SHERWOOD et al., 2009). O *FlowVisor* consegue criar camadas virtuais de rede e delegar o controle de cada camada para um controlador *OpenFlow* diferente.

As metodologias propostas por Natarajan, Huang e Wolf são: *Hash-Trie based Conflict Detection* (HTCD) e *Ontology based Conflict Detection*(OCD).

A técnica HTCD consiste em um algoritmo do paradigma *"divide and conquer"* para detectar conflitos. Nessa abordagem, os campos dos fluxos *OpenFlow* com *wildcards* (coringas) e valores absolutos são divididos utilizados para a criação de uma tabela *hash* onde será criada uma matriz de intersecção baseado no *hash* dos campos do fluxo *OpenFlow*. No processo de combinação o algoritmo então combina todos os campos e com base na matriz de intersecção determina quais fluxos estão em conflito.

A técnica OCD fornece um mecanismo baseado em ontologia para detecção de conflitos. Um sistema de lógica baseado em ontologia fornece uma mecanismo padrão para a representação do conhecimento e raciocínio automático (inferência) com uma semântica e sintaxe bem definida. Cada fluxo é convertido em uma representação lógica utilizando o mecanismo de descrição lógica, e depois as representações são enviadas a um sistema de inferência para determinar os conflitos existentes.

Al-Shaer e Al-Haj propuseram uma ferramenta de verificação para encontrar error de configuração na tabela de fluxos utilizando Diagramas de Decisão Binário (BDD) chamado de *FlowChecker*. Esta abordagem cria uma representação da tabela de fluxo de todos os *switches* conectados a um determinado controlador *OpenFlow*, validando a corretude da configuração das tabelas de fluxo(AL-SHAER; AL-HAJ, 2010).

Diferente dos outros trabalhos, este propõe-se a criar uma forma de representar o conflito de fluxos utilizando lógica de primeira ordem e aplicar essas regras em um agente lógico que irá ser executado em uma *engine Prolog* como processo residente em um controlador *OpenFlow*. Esse processo residente poderá encontrar conflitos em tempo de execução, economizando recursos dos ativos de rede e minimizando erros de configuração em uma rede *OpenFlow*. Diferente das técnicas utilizadas por Natarajan, Huang e Wolf e por Al-Shaer e Al-Haj, o presente trabalho implementará um software que utilizará técnicas de inteligência artificial para alcançar esse objetivo.

1.3 Contribuições

Durante o processo de pesquisa deste trabalho, alguns artigos relacionados com a proposta, foram publicados no decorrer do processo.

Inicialmente, foi publicado um artigo descrevendo uma arquitetura de um Gerenciador de Redes *OpenFlow* Baseado em Políticas (PBONM) (BATISTA; CAMPOS; FERNANDEZ, 2013) onde nessa arquitetura é proposta um sistema que forneça uma linguagem baseada em política para definição de fluxos *OpenFlow*.

A arquitetura do PBOMN define que a linguagem utilizada pada definição de fluxos será uma variante do *Ponder* (SLOMAN, 1994). Um artigo, com a definição da variante da linguagem *Ponder* chamada de *PonderFlow*, foi publicado (BATISTA; FERNANDEZ, 2014). Nesse artigo já era previstoum analisador de fluxos conflitantes que converteria os fluxos definidos em *PonderFlow* para uma representação intermediária para análise de conflitos.

Um terceiro artigo foi publicado descrevendo o componente de análise de conflitos presente neste trabalho (BATISTA; CAMPOS; FERNANDEZ, 2014). Este componente é uma implementação concreta da abstração do agente lógico proposto por Russel&Norving (RUSSELL; NORVIG, 2010) e por Wooldridge (WOOLDRIDGE, 2001) utilizando a linguagem de programação *Prolog* para definição das regras que definem um conflito e implementação do agente.

1.4 Estrutura do Trabalho

Este trabalho encontra-se organizado da seguinte forma: o Capítulo 2 apresenta os fundamentos conceituais, mostrando um resumo sobre *OpenFlow*, lógica de primeira ordem, *Prolog* e a definição de agente lógico; o Capítulo 3 apresenta toda a metodologia de desenvolvimento do mecanismo de detecção de conflitos para a arquitetura *OpenFlow*; o Capítulo 4 mostra o ambiente de emulação, ferramentas utilizadas e o modelo do agente lógico de detecção de conflitos implementado; o Capítulo 5 mostra os resultados obtidos com o mecanismo de detecção de conflitos ; e finalmente o Capítulo 6 encerra o trabalho com as conclusões obtidas e sugestões de trabalhos futuros.

2 Fundamentos Conceituais

Nesse capítulo, será apresentada uma introdução aos conceitos teóricos utilizados nesse trabalho. Na seção 2.1, será apresentada uma introdução sobre a arquitetura OpenFlow e o seu funcionamento. A seguir, na seção 2.2, será apresentado um resumo sobre lógica de primeira ordem e alguns exemplos sua utilização. Em seguida, será apresentado, na seção 2.3, uma introdução à linguagem de programação Prolog e, finalmente, na seção 2.4, uma introdução ao conceito de agente lógico e de sua arquitetura geral.

2.1 OpenFlow

O protocolo *OpenFlow* é uma proposta da iniciativa *Clean Slate* que define uma protocolo aberto e padronizado para configurar as tabelas de encaminhamento de elementos de rede, tais como *switches* e roteadores (MCKEOWN et al., 2008).

O *OpenFlow* é a base das redes definidas por software (SDN), permitindo que um administrador de redes modifique o comportamento de uma rede de computadores dinamicamente, além de desacoplar o plano de controle do plano de dados.

Os componentes que compõe o *OpenFlow* são:

- Um ou mais elementos de rede *OpenFlow*;
- O controlador *OpenFlow*;
- O protocolo *OpenFlow*.

O controlador *OpenFlow* é uma entidade centralizada que configura e monitora todos os dispositivos da rede. Além disso o controlador fornece duas interfaces de programação para aplicações (API) chamadas *Northbound API* e *Southbound API*.

A *Northbound API* permite que aplicações voltadas às redes de computadores possam tirar proveito do poder que o controlador *OpenFlow* pode oferecer como, por exemplo, configurar uma rede *OpenFlow* de acordo com o resultado de um algoritmo que reconheceu uma anomalia no trafego de rede.

A *Southbound API* permite que o controlador *OpenFlow* possa se conectar aos elementos de rede *OpenFlow* através de um protocolo padronizado, chamado de protocolo *OpenFlow* que será detalhado mais adiante.

Atualmente existem diversos controladores *OpenFlow* disponíveis gratuitamente, onde podemos citar o HermesNet (FERNANDEZ, 2014), Floodlight (ERICKSON, 2014), Pox (NOXRepo.org, 2014), Trema (NEC Corporation, 2014), entre outros.

Um elemento de rede *OpenFlow* é qualquer elemento de rede existente na rede como, por exemplo, *switches*, roteadores ou pontos de acesso sem fio que implementem o protocolo *OpenFlow*. Cada elemento de rede possui uma tabela de fluxos que indica a ação a ser aplicada a um ou mais pacotes de um determinado fluxo de rede (MCKEOWN et al., 2008).

Existem atualmente dois tipos de dispositivos *OpenFlow*: Elementos de redes *OpenFlow* dedicado e os elementos de rede habilitados ao *OpenFlow* (MCKEOWN et al., 2008).

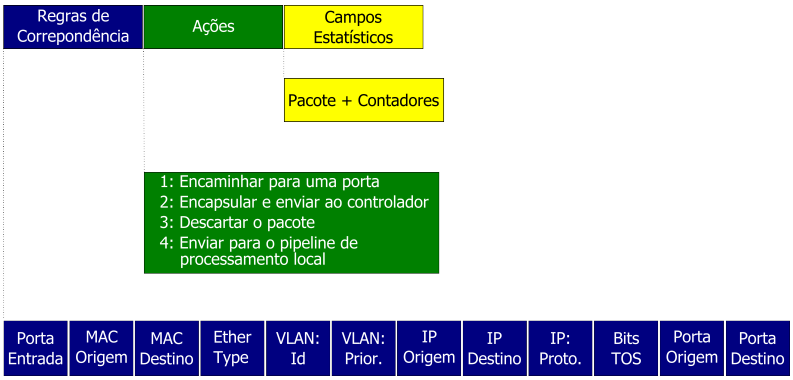
Um elemento de rede *OpenFlow* dedicado é um dispositivo "burro"que apenas consegue realizar o encaminhamento de pacotes, mas não consegue definir o destino de um fluxo de rede recém criado. Nesse caso o elemento de rede depende do controlador para tratar os novos fluxos advindos da rede *OpenFlow*.

Os elementos de rede habilitados ao *OpenFlow*, diferentemente dos elementos de rede *OpenFlow* dedicado, possuem a habilidade de atuar como um elemento de rede *OpenFlow* ou atuar como um elemento de rede tradicional. Isso ocorre pois esses tipos de elementos de redes são dispositivos comuns onde o fabricante apenas adicionou no *firmware* do equipamento a capacidade do mesmo de operar em redes *OpenFlow*.

O protocolo *OpenFlow* atua como uma interface entre o controlador *OpenFlow* e os dispositivos de rede (*Southbound API*). A partir desse protocolo é possível configurar as tabelas de fluxos dos elementos de rede *OpenFlow* adicionando ou removendo entradas de fluxo *OpenFlow*.

Uma entrada de fluxo *OpenFlow* possui três partes: Os campos com as regras de correspondência, um conjunto de ações e os campos estatísticos, conforme a Figura 1.

Figura 1 – Entrada de fluxo OpenFlow

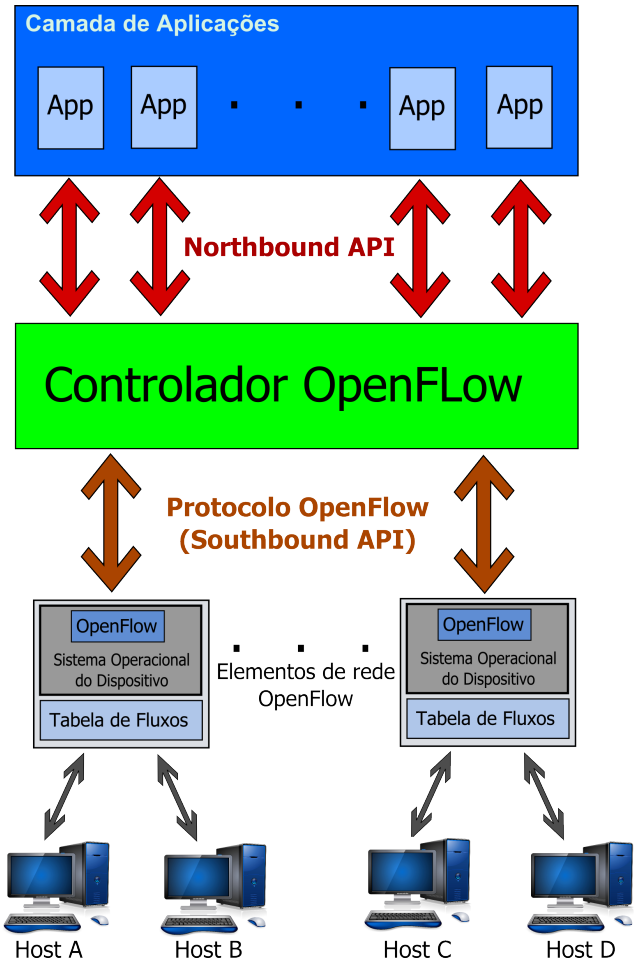


Os campos com as regras de correspondência é um 12-tupla (versão 1.0 do *OpenFlow*) usados para definir as condições de correspondência de um ou mais fluxos dependendo da sua definição. O conjunto de ações especifica quais ações serão executadas nos fluxos que corresponderem a uma entrada de fluxo. Finalmente, os campos estatísticos são utilizados

para contar a ocorrência dos fluxos, quantidade de pacotes recebidos, quantidade de bytes recebidos etc.

A Figura 2 fornece uma visão de todos os componentes de uma rede *OpenFlow* e de como os mesmos se relacionam entre si.

Figura 2 – Arquitetura OpenFlow



Em uma rede *OpenFlow*, quando um pacote qualquer é enviado a uma porta de um elemento de rede *OpenFlow*, o mesmo irá procurar na sua tabela de fluxos alguma entrada que corresponda ao pacote recebido. Caso exista alguma entrada em que ocorra uma correspondência, então o elemento de rede irá executar a ação associada a mesma.

Se não ocorrer uma correspondência com nenhuma entrada na tabela de fluxos, o elemento de rede encapsula o pacote em uma mensagem do protocolo *OpenFlow* e envia para o controlador realizar uma análise do mesmo. O controlador, por sua vez, desencapsula o pacote, analisa seu conteúdo, escreve na tabela de fluxos do dispositivo solicitante via protocolo *OpenFlow* o que deve ser realizado com esse pacote e os seus subsequentes e, em seguida, encaminha o pacote novamente para o elemento de rede solicitante.

Ao regressar, o pacote é analisado novamente pelo elemento de rede e, como este agora possui uma entrada que corresponda ao pacote graças a intervenção do controlador,

então executa a ação associada a mais nova entrada da tabela de fluxos. As possíveis ações podem ser: Encaminhar o pacote para uma outra porta do dispositivo; Encapsular e enviar para o controlador; Descartar o pacote; Enviar para o *pipeline* de processamento local (ação disponível nos dispositivos habilitados ao *OpenFlow*).

2.2 Lógica de Primeira Ordem

A lógica de primeira ordem, ou cálculo de predicados, é um sistema lógico que estende a lógica proposicional aumentando o seu poder de expressão.

A lógica proposicional possui limitações com respeito a codificação de sentenças declarativas. De fato, o cálculo proposicional manipula de forma satisfatória componentes da sentença como não, e, ou, se... então, mas certos aspectos lógicos que aparecem em linguagens naturais ou artificiais são muito mais ricos (MOURA, 2014).

Considere a seguinte sentença declarativa: *Todo estudante é mais jovem do que algum instrutor*.

Na lógica proposicional, pode-se identificar esta sentença com uma variável proposicional p. No entanto, esta codificação não reflete os detalhes da estrutura lógica desta sentença, tais como:

- Ser um estudante
- Ser um instrutor
- Ser mais jovem do que alguém

Para expressar estas propriedades, utilizam-se predicados. Por exemplo, podemos escrever *estudante(ana)* para denotar que *Ana é uma estudante*. Da mesma maneira pode-se escrever *instrutor(marcos)* para denotar que *Marcos é instrutor*. Por fim, pode-se escrever *jovem(ana,marcos)*, para denotar que *Ana é mais jovem que Marcos*. Nestes exemplos, *estudante*, *instrutor* e *jovem* são exemplos de predicados (MOURA, 2014).

Ainda necessitam-se definir as noções de *"todo"* e *"algum"*. Para isto, introduziu-se o conceito de variável. Variáveis serão denotadas por letras latinas minúsculas do final do alfabeto: *u,v,w,x,y,z* (possivelmente, acrescidas de sub-índices $x_1, x_2, ...x_n$) (MOURA, 2014).

Utilizando variáveis, pode-se especificar o significado dos predicados estudante, instrutuor e jovem de uma maneira mais forma:

- *estudante(x)*: x é um estudante.
- *instrutor(y)*: y é um instrutor.

- *jovem(x,y)*: x é mais jovem que y.

Para expressar detalhes a sentença *Todo estudante é mais jovem do que algum instrutor*, precisa-se codificar o significado de **todo** e **algum**. Os quantificadores ∀ e ∃ realizam esse trabalho:

- ∀: significa *para todo*;
- ∃: significa *existe*.

Os quantificadores ∀ e ∃ estão sempre ligados a alguma variável:

- $\forall_x$: significa *para todo x*;
- $\exists_y$: significa *existe um y* ou *existe algum y*.

Diante disso pode-se codificar a sentença *Todo estudante é mais jovem do que algum instrutor* da seguinte forma:

$$\forall_x(estudante(x) \rightarrow (\exists_y(intrutor(y) \wedge jovem(x,y))))$$

2.3 Prolog

As linguagens de programação procedimentais e orientadas a objetos são utilizadas passando-se instruções, que executadas, uma a uma, em uma determinada ordem, permite que um computador resolva um determinado problema. A linguagem de programação *Prolog* não segue esta regra.

Um programa *Prolog* consiste em um conjunto de fatos que juntamente com um conjunto de condições, nas quais satisfazendo-as, o programa alcança a resolução de um determinado problema. Dessa forma um computador "realiza deduções"baseado-se nos fatos fornecidos à solução do problema (COVINGTON; NUTE; VELLINO, 1997).

Prolog é baseada na lógica formal da mesma maneira que *FORTRAN* e *BASIC*, é similar a linguagens baseadas em aritmética e álgebra simples. *Prolog* resolve problemas aplicando técnicas originalmente desenvolvidas para provar teoremas lógicos.

Além de ser uma linguagem muito versátil, *Prolog* foi inventada por Alain Colmerauer e seus colegas da *University of Aix-Marseille*, França, em 1972. O nome significa *PROgramming in LOGic*. A linguagem de programação *Prolog* é utilizada principalmente em aplicações de inteligência artificial, especialmente em sistemas automatizados de raciocínio(COVINGTON; NUTE; VELLINO, 1997).

Prolog possui muitas semelhanças com *Lisp*, a linguagem de programação tradicionalmente usada para pesquisa em inteligência artificial. Ambas as linguagem tornam fáceis

a computação em dados complexos, e ambas possuem o poder de expressar algoritmos elegantemente.

Ambas alocam memória dinamicamente, o que torna desnecessária a declaração do tamanho das estruturas de dados antes de cria-las. Ambas a linguagem permitem que o programa examine e modifique a si próprio, tornando possível que um programa "aprenda"através das informações obtidas em tempo de execução.

A principal diferença entre o *Prolog* e o *Lisp* é que o *Prolog* possui um procedimento de raciocínio automatizado chamado de *mecanismo de inferência*, enquanto que *Lisp* não possui tal mecanismo. Como resultado, programas que necessitam de raciocínio lógico são muito mais fáceis de escrever utilizando-se *Prolog* do que em *Lisp* (COVINGTON; NUTE; VELLINO, 1997).

2.4 Agente Lógico

O *Agente Lógico* é um programa composto por declarações lógicas e fatos que, através delas, o mesmo consegue raciocinar para obter uma solução.

O componente central de um *Agente Lógico* é a *base de conhecimento*. Uma base de conhecimento nada mais é do que um conjunto de sentenças e cada sentença é expressa em uma linguagem chamada de *Linguagem de Representação do Conhecimento*, representando algumas asserções sobre o mundo de discurso.

Deve ser possível adicionarem-se novas sentenças à base de conhecimento e consultar o que se conhece. Ambas as tarefas podem envolver inferência (derivação de novas sentenças a partir das antigas)(RUSSELL; NORVIG, 2010).

O processo de execução de um *Agente Lógico* baseia-se em:

1. Informar a base de conhecimento o que o agente esta percebendo do ambiente;
2. Pergunta a base de conhecimento qual a próxima ação que deve ser executada. Um extensivo processo de *raciocínio lógico* é realizado sobre o base de conhecimento para que sejam decididas as ações que devem ser executadas;
3. Realiza a ação escolhida e informa a base de conhecimento sobre a ação que está sendo realizada.

Geralmente, utiliza-se uma linguagem lógica de representação do conhecimento, e.g *Prolog*, pois:

- **Facilita a criação dos agentes**,pois é possível dizer o que o agente sabe através de sentenças lógicas;

- O agente pode **adicionar** novas sentenças a sua base de conhecimento enquanto ele explora o ambiente;
- Abordagem **declarativa** de criação de sistemas inteligentes.

3 Metodologia

Será abordada nesse capítulo, a metodologia da concepção do mecanismo de detecção de conflitos e, conseqüentemente, do agente lógico, descrevendo cada etapa do processo de concepção criação e implementação. Inicialmente a formalização matemática da tabela de fluxos *OpenFlow* será apresentada e, em seguida, os conflitos serão abordados e serão classificados e definidos formalmente.

Finalmente, de posse de toda formulação matemática necessária, a arquitetura do agente lógico proposto será apresentada, além dos detalhes da implementação *Prolog*.

3.1 Definição Matemática da Tabela de Fluxos

A tabela de fluxos *OpenFlow*, conforme a Figura 1, é composta por campos de correspondência (*matching fields*), uma ou mais ações e pelos campos estatísticos.

O trabalho de Al-Shaer e Al-Haj (AL-SHAER; AL-HAJ, 2010) introduziu uma formalização matemática para a tabela de fluxos *OpenFlow*. Na formalização original, os autores modelaram matematicamente o tratamento de várias tabelas de fluxos, onde vários usuários podem manipular os fluxos uma mesma tabela.

Entretanto o presente estudo modificou a formalização proposta por Al-Shaer e Al-Haj com intuito de trabalhar apenas com uma única tabela de fluxos onde apenas um único usuário será capaz de manipula-la, simplificando o processo de criação do agente lógico.

Outra consideração a ser feita é que o protocolo *OpenFlow* prevê a existência de um campo de prioridade, caso dois ou mais fluxos atendam simultaneamente um mesmo pacote de rede. Nesse estudo, qualquer fluxo o campo prioridade sempre possuirá o mesmo valor, o que ocasionará uma precedência natural em relação a ordem de inserção na tabela de fluxos, ou seja, o primeiro fluxo tem prioridade sobre o segundo, o segundo sobre o terceiro e assim sucessivamente.

A seguir algumas definições serão apresentadas para formalizar a tabela de fluxos *OpenFlow*.

Definição 1. *Uma tabela de fluxos* Γ *é uma seqüencia de entradas de fluxos* F_n *que determinam o comportamento de um swicth OpenFlow, ou seja:*

$$\Gamma = F_1, F_2, \ldots, F_n \tag{3.1}$$

Definição 2. *Uma entrada de fluxo* F_n *é um conjunto de* k *campos de comparação* $F = f_1, f_2, \ldots, f_n$ *onde* n *é a quantidade de campos de entrada, além do fluxo* F *estar*

associado a uma ação a_i pertencente ao conjunto de ações possíveis. Assim um fluxo de entrada pode ser definido como:

$$F_n := C_i \rightarrow a_i \in A \tag{3.2}$$

Onde C_i são as restrições dos campos de comparação que devem ser satisfeitos para que a ação $a_i \in A$ possa ser executada. A condição C_i pode ser representada como uma expressão booleana sobre os valores dos campos de comparação $fv_1, fv_2, \ldots, fv_n$, como mostrado em 3.3:

$$C_i = fv_1 \wedge fv_2 \wedge \ldots \wedge fv_n \tag{3.3}$$

Como dito anteriormente, quando duas ou mais entradas de fluxo possuem a mesma prioridade, a primeira correspondência será utilizada, ou seja:

$$\Upsilon_a = \bigvee_{i \in index(a)} (\neg C_1 \wedge \neg C_2 \wedge \ldots \wedge C_i) \Rightarrow$$
$$\Upsilon_a = \bigvee_{i \in index(a)} \bigwedge_{j=1}^{i-1} \neg C_j \wedge C_i$$
$$tal\ que,\ prioridade(C_{i-1}) < prioridade(C_i)$$
$$\text{e } index(a) = \{i \mid F_i = C_i \rightarrow a\}$$

Onde Υ_a é o conjunto de todos os fluxos que possuem a ação a pertencente ao conjunto de ações A como ação a ser executada caso um pacote de rede corresponda aos critérios do fluxo.

Assim os fluxos *OpenFlow* de um *switch* j que executam uma ação a qualquer, podem ser representados conforme a equação 3.4:

$$\Gamma(j) = \bigvee_{\forall\ n\ \in\ index(a)} \Upsilon_n \tag{3.4}$$

3.2 Definição de Conflitos

Os conflitos na tabela de fluxos *OpenFlow* serão abordados nessa sessão. Podem-se classificar os conflitos em dois tipos distintos: conflitos estáticos e conflitos dinâmicos. Cada tipo de conflito será tratado nas subseções a seguir.

3.2.1 Conflitos Estáticos

Um conflito estático surge a partir de um fluxo semânticamente mal formado durante o processo de definição do mesmo por um usuário ou um administrador de rede. Os controladores *OpenFlow* não fornecem nenhum mecanismo de verificação para esse tipo de conflito. Abaixo são apresentados alguns exemplos conflitos estáticos:

1. **Endereço *MAC* de origem igual ao endereço *MAC* de destino:** Um fluxo *OpenFlow* no qual durante o processo de correspondência de pacotes de rede, teria correspondência direta a um pacote de rede onde o endereço *MAC* de origem é exatamente igual ao endereço *MAC* de destino.
2. **Endereço *IP* de origem igual ao endereço *IP* de destino:** Um fluxo *OpenFlow* no qual durante o processo de correspondência de pacotes de rede, teria correspondência direta a um pacote de rede onde o endereço *IP* de origem é exatamente igual ao endereço *IP* de destino.

Os conflitos acima descritos, mesmo não impactando no comportamento de uma rede *OpenFlow*, consomem recursos do *switch* em que foram definidos.

Assim podem-se definir os conflitos estáticos como um conjunto de todos os conflitos conhecidos, ou seja:

$$\Phi = \{\phi_1, \phi_2, \ldots, \phi_n\}, n \in \mathbb{N} \tag{3.5}$$

Cada conflito em 3.5 pode ser representado como uma declaração booleana das restrições dos campos de comparação.

Tomando o primeiro exemplo de conflito estático que fora descrito nesta subseção (endereço MAC de origem igual ao endereço MAC de destino), é possível expressa-lo da seguinte forma:

$$\phi = \forall F_i \in \Gamma(j), fv_2 = fv_3 \tag{3.6}$$

Ou seja, para qualquer entrada de fluxo F_i pertencente a tabela de fluxos Γ de um *switch OpenFlow* j da rede, onde o valor do segundo campo for igual ao valor do terceiro campo ($fv2 = fv3$, que correspondem ao endereço *MAC* de origem e destino respectivamente) caracteriza-se um conflito estático.

3.2.2 Conflitos Dinâmicos

Os conflitos dinâmicos ocorrem quando dois fluxos *OpenFlow* semânticamente corretos, definidos em uma mesma tabela de fluxos de um *switch OpenFlow*, podem tratar

simultaneamente de um mesmo de pacote, ou seja, durante o processo de correspondência ambos os fluxos podem tratar este pacote.

Entretanto, o conjunto de ações definido no primeiro fluxo, dependendo da ordem de inserção dos fluxos, será o conjunto de ações a ser executado após o processo de correspondência. Para exemplificar, tome como exemplo a seguinte estrutura de uma rede *OpenFlow* apresentada na Figura 3 para exemplificar o conflito dinâmico.

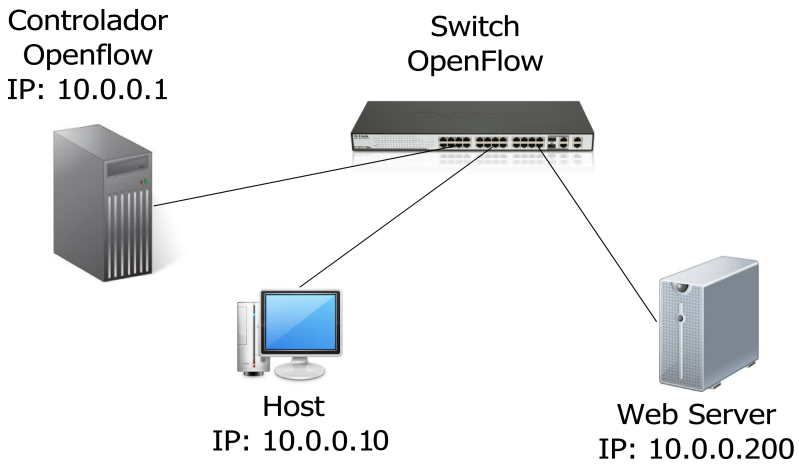


Figura 3 – Exemplo de uma rede OpenFlow

No exemplo da rede *OpenFlow* ilustrado na figura acima, temos um *host* com endereço *IP* 10.0.0.10 e um servidor Web com endereço *IP* com endereço 10.0.0.200. Agora, imagine que um administrador de redes adicionou os seguintes fluxos na tabela de fluxos do *swicth OpenFlow*:

1. **Negar qualquer solicitação à porta 80 do servidor web originado na rede 10.0.0.0:** Nenhum computador da rede local conseguirá acessar o servidor via protocolo HTTP (que possui a porta 80 como porta padrão).
2. **Permitir que o computador cujo o endereço *IP* de origem 10.0.0.10 acesse o computador cujo o endereço *IP* seja 10.0.0.200:** O *host* possui permissão para acessar o servidor Web independente da porta de destino, inclusive a porta 80.

Ainda, suponha que o Fluxo 1 possua *DROP* (descartar o pacote de rede) como ação a ser executada, caso algum pacote corresponda, positivamente, ao fluxo 1, e o fluxo 2 possua *OUTPUT:X* (encaminhar o pacote para a porta X do *switch*, nesse caso a porta na qual o servidor Web está conectado ao *switch*) como ação a ser executada, caso o pacote corresponda, positivamente, ao fluxo 2.

Fica evidente que quando o *host* cujo o endereço *IP* de origem é 10.0.0.10 (endereço pertencente a rede 10.0.0.0) enviar uma solicitação para o servidor Web, cujo o endereço *IP* de destino é 10.0.0.200, utilizando a porta 80 como porta de destino ambos os fluxos poderão ser utilizados no processo de correspondência (*matching*) desse pacote.

Entretanto dependendo da ordem de inclusão dos fluxos na tabela de fluxos pode ocorrer um dos seguintes casos:

- Se o fluxo 1 for inserido primeiro que o fluxo 2, então o *host* não acessará o serviço web disponibilizado pelo servidor. Significa dizer que o fluxo 2 nunca será utilizado no processo de correspondência (*matching*) nesse caso.

- Se o fluxo 2 for inserido primeiro que o fluxo 1, então o *host* poderá acessar o serviço web disponibilizado pelo servidor. Significa dizer que o fluxo 1 nunca será utilizado no processo de correspondência (*matching*) nesse caso.

Dependendo do contexto, um dos casos supra citados, pode ser classificado como um conflito dinâmico. Entretanto, nenhum controlador *OpenFlow* possui algum mecanismo de alerta para esse tipo de conflito.

Assim, podemos definir o conflito dinâmico como um conjunto de todos os conflitos conhecidos, ou seja:

$$\Psi = \{\psi_1, \psi_2, \ldots, \psi_n\}, n \in \mathbb{N} \tag{3.7}$$

Suponha que $P = p_1, p_2, \ldots, p_n$ seja um pacote de rede que acabou de entrar por alguma porta de um *switch OpenFlow*, sendo que p_n corresponde ao campos do cabeçalho do pacote. Um conflito dinâmico pode ser definido como:

$$\begin{aligned} \psi = & \forall F_a, F_b \in \Gamma(j), \\ & \text{e P é o pacote recebido,} \\ & \text{e } \forall fv_k^a = * \text{ então } fv_k^a = p_k, \\ & \text{e } \forall fv_k^b = * \text{ então } fv_k^b = p_k \end{aligned} \tag{3.8}$$

Ou seja, dado uma tabela de fluxos de um *switch OpenFlow* tomando suas entradas de fluxos dois a dois, no caso F_i e $F_(i+1)$. Se existir algum coringa (*wildcard*) na definição do fluxo de entrada (o coringa * que significa qualquer valor), o coringa será substituído pelo valor correspondente do cabeçalho *OpenFlow* do pacote P.

Após isso, seria realizado uma verificação campo a campo nos fluxos F_i e $F_(i+1)$, se todos os campos forem iguais, então existe um conflito estático, ou seja:

$$fv_1^i = fv_1^{i+1} \wedge fv_2^i = fv_2^{i+1} \wedge \ldots \wedge fv_k^i = fv_k^{i+1} \tag{3.9}$$

3.3 Agente Lógico Baseado no Modelo Reflexivo

O agente lógico baseado no modelo reflexivo é um tipo de agente inteligente que será utilizado para detectar fluxos conflitantes onde utilizará regras de condição-ação juntamente com lógica de primeira ordem.

A seguir, será descrito o agente reflexivo baseado em modelos com regras de condição-ação e em seguida a integração da lógica de primeira ordem no agente proposto.

3.3.1 Agente Reflexivo Baseado em Modelos com Regras de Condição-Ação

A estrutura desse agente é um agente reflexivo simples com regras de condição-ação (RUSSELL; NORVIG, 2010), com a inserção da *função próximo* usada para adaptar o estado interno para lidar com ambientes parcialmente observáveis.

Esta nova informação descreve aspectos do ambiente (chamado de modelo) que não são atualmente perceptíveis pelos sensores do agente.

Especificamente, a função *próximo estado* se adapta ao estado interno atual do agente considerando as informações da percepção e da informação sobre os efeitos de possíveis ações no ambiente e sobre como o ambiente evolui de forma independente das ações do agente.

A Figura 4 mostra o diagrama do agente reflexivo baseado em modelo, sintetizado pelas ideias de Russell &Norvig, relacionadas no programa agente reativo (RUSSELL; NORVIG, 2010), bem como a arquitetura abstrata do ponto de vista proposto por Wooldridge (WOOLDRIDGE, 2001).

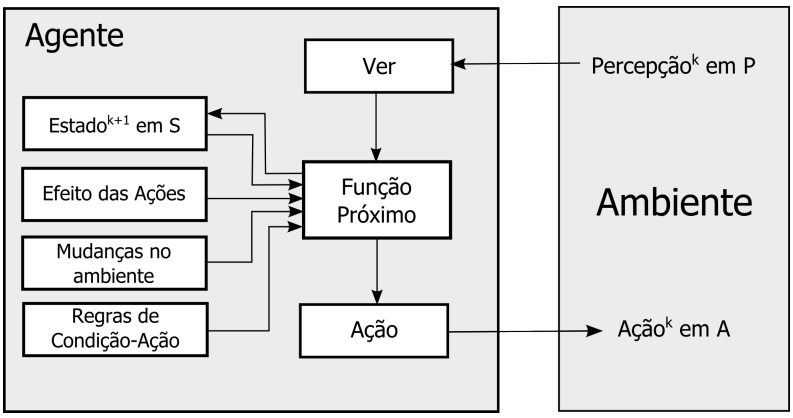


Figura 4 – Diagrama esquemático do Agente Reflexivo Baseado em Modelo

Em adição à descrição funcional realizada em (RUSSELL; NORVIG, 2010) ao agente reflexivo simples, a descrição dessa nova estrutura adiciona um passo intermediário no esquema de cinco etapas. Essa nova síntese assume que para qualquer interação k:

1. Através dos sensores o agente recebe a informação do ambiente (*Env*), i.e, as

percepções definidas em um conjunto de n percepções potenciais a partir do ambiente, $P = p_1, p_2, \ldots, p_n$;

2. O subsistema de percepções, *Ver*:$P \rightarrow P'$, processa cada percepção em P e mapeia para um possível estado P', que é uma representação dos aspectos na informação de percepção que não está acessível para o agente;

3. A função de próximo estado, *próximo*:$P'xS \rightarrow S$, mapeia os estados em P' e o atual estado interno em um provável novo estado interno em S, considerando os efeitos das ações e analisando as mudanças do ambiente, $S = s1, \ldots, s_m$;

4. O subsistem de tomada de decisão, *ação*:$S \rightarrow A$, processa o estado interno em S e seleciona, de acordo com uma regra específica do conjunto de regras condição-ação, uma das ações dentro do conjunto de ações possíveis definida para o agente, $A = a1, \ldots, an$;

5. Através dos atuadores, o agente envia a ação selecionada para o ambiente;

6. Na interação $k + 1$, o agente inicia outra ciclo envolvendo a percepção do mundo através da função *Ver*, a atualização do seu estado interno através da função *próximo* e a seleção de uma nova ação através da função *ação*.

Nesse esquema, as regras de condição-ação podem ser vistas como um conjunto de associações comuns, nas quais são observadas entre condições específicas estabelecidas das descrições dos estados interno em S e de certas ações possíveis em A.

O projetista do agente define essas regras já se preocupando com a medida de desempenho que irá ser aplicada ao agente. Nesse contexto, espera-se que, num determinado ambiente, se as regras estão apropriadas, o agente irá atingir seus objetivos e, conseqüentemente, obterá uma bom desempenho.

3.3.2 Lógica de Primeira Ordem aplicada ao Agente Reflexivo Baseado em Modelos

Conforme explicado anteriormente, um conflito estático surge no momento de sua criação, quando defini-se um fluxo semanticamente errôneo.

Um tipo de conflito estático que o agente lógico deve detectar é o caso de uma entrada de fluxo *OpenFlow* possuir o endereço MAC de origem igual ao endereço MAC de destino, pois esse tipo de pacote deverá ser tratado localmente no *host* de origem e nunca chegará ao *switch OpenFlow*.

Então a condição primária da regra de condição-ação na definição *Prolog* o predicado *fazer/1* foi declarado para realizar a detecção esse tipo de conflito, i.e:

Listagem 3.1 – Predicado fazer/1 para conflitos estáticos

```
1 fazer(conflito(estatico1,ID)):-
    fluxo(ID,_,MacSrc,MacDst,_,_,_,_,_,_,_,_,_),
3   MacSrc == MacDst.
```

Entretanto um conflito dinâmico deve ser detectado pelo agente quando dois fluxos distintos podem, simultaneamente, tratar o mesmo pacote de rede. Para detectar esse tipo de conflito, a abordagem proposta por esse trabalho emprega um predicado dinâmico chamado *pacote/12* para representar qualquer pacote de rede.

Então a declaração da regra de condição-ação para o predicado *fazer/1* foi declarado como:

Listagem 3.2 – Predicado fazer/1 para conflitos dinâmicos

```
1 fazer(conflito(dinamico1,[ID1, ID2])):-
    pacote(X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12),
3   fluxo(ID1,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12),
    fluxo(ID2,X1,X2,X3,X4,X5,X6,X7,X8,X9,X10,X11,X12),
5   diferente(ID1,ID2)
```

Onde o predicado *diferente/2* é um predicado computável para verificar se um par de fluxos possuem os identificadores diferentes, isto é, se ID1 une-se com ID2, além de verificar se o conflito detectado entre o par de fluxos não foi considerado anteriormente no processo de detecção de conflito.

4 Implementação da Proposta

Nesta seção, serão abordados os principais detalhes do processo de implementação do agente lógico e da base de conhecimento utilizada para a validação do mesmo.

Para que o agente seja considerado racional, o mesmo deve detectar os seguintes tipos de conflitos baseando-se nas definições do predicado *do/1* exemplificadas na subseção 3.3.2:

1. Endereço MAC de origem igual ao endereço MAC de destino (conflito estático);
2. Endereço IP de origem igual ao endereço IP de destino (conflito estático);
3. Dois fluxos que não são mutuamente excludentes, tomando-se como exemplo o caso da subseção 3.2.2 (conflito dinâmico).

A base de conhecimento agente foi definida, inicialmente, com sete fluxos OpenFlow onde os quatro primeiros não apresentam nenhum tipo de conflito e os três restantes apresentam algum tipo de conflito.

O fluxo de ID 5 exemplifica um fluxo que possui um conflito estático onde o endereço MAC de origem é igual ao endereço MAC de destino. O fluxo de ID 6 exemplifica um fluxo que possui um conflito estático onde o endereço IP de origem é igual ao endereço IP de destino.

E o fluxo com ID 7 exemplifica um fluxo que possui um conflito dinâmico com o fluxo com ID 4. Esses dois fluxos, dependendo do tipo de pacote, podem simultaneamente serem usados no processo de comparação (*matching*). Entretanto, por ter sido declarado na tabela de fluxos primeiro, a ação do fluxo 4 será executada por possui maior prioridade sobre o fluxo 7. A Listagem 4.1 exemplifica a tabela de fluxos utilizada na base de conhecimento do agente.

Listagem 4.1 – Tabela de Fluxos utilizada para validar o agente lógico

```
1 flow(1,_,_,'FF:FF:FF:FF:FF:FF','0x0800',_,_,_,_,_,_,_,_).
  flow(2,_,_,_,'0x0800',_,_,'10.0.0.1','10.0.0.2',_,_,_,_).
3 flow(3,_,_,_,'0x0800',_,_,'10.0.0.2','10.0.0.1',_,_,_,_).
  flow(4,_,_,_,'0x0800',_,_,_,'10.0.0.10',tcp,_,_,80).
5 flow(5,_,'78:12:DA:A8:CB:C1','78:12:DA:A8:CB:C1','0x0800',_,_,_,_,_,_,_,_).
  flow(6,_,_,_,'0x0800',_,_,'10.0.0.3','10.0.0.3',_,_,_,_).
7 flow(7,_,_,_,'0x0800',_,_,'10.0.0.1',_,tcp,_,_,_).
```

Inicialmente o agente lógico começa com o predicado *detector/0*, onde começará todo o processo de detecção de conflitos. Dentro desse predicado há a chamada de outros três predicados: (1) *sensor/2*, (2) *detetor/2* e (3) *atuador/2*;

O predicado *sensor/2* realiza uma leitura das políticas (que são predicados *policy/6* que serão inseridas na base de conhecimento do agente) e armazena o resultado em uma lista de políticas P, ou seja, representando as informações perceptivas do detector, que são alimentadas ao programa agente através do predicado *detetor/2*.

O predicado *detetor/2*, como mostrado na Listagem 4.2, realiza o mapeamento $P \rightarrow A$. Em vias de fato, é uma concretização da arquitetura abstrata do agente com estado interno descrito por Wooldridge (WOOLDRIDGE, 2001) e do programa agente *Simple Reflex Model-Based* presente em Russel&Norving (RUSSELL; NORVIG, 2010).

Listagem 4.2 – Predicado detetor/2

```
1 detetor(P,A):-
    ver(P,S),
3   proximo(S),
    acao(A).
```

O predicado *proximo/1* insere na base de conhecimento do agente cada uma das políticas na lista de políticas S. Enquanto que o predicado *acao/1* implementa o mecanismo de tomada de decisão do agente que é baseado em regras de condição-ação.

Junto ao predicado *acao/1* foi necessário criar um novo predicado *faca_todas/1*, capaz de verificar todas as regras condição-ação do agente e retornar as ações daquelas regras com as condições ativadas, ou seja, retornar uma lista de ações A.

O código fonte completo do agente proposto nesse trabalho está disponível no apêndice A sob a licença GPL v3.

5 Resultados

Nesta seção serão abordados os resultados obtidos dessa pesquisa, além de resultados sobre a performance do agente lógico.

Foram realizados testes para avaliar a sensibilidade do agente na detecção de conflitos, os dados que foram inseridos na base de conhecimento do agente, mostrados na listagem 4.1, foram duplicados para efeitos de testes. A tabela 1 resume os resultados para o caso supra citado.

Tabela 1 – Conflitos Detectados

Conflict	**ID 1**	**ID 2**
estatico1	5	_
estatico1	12	_
estatico2	6	_
estatico2	13	_
dinamico1	4	7
dinamico1	4	11
dinamico1	4	14
dinamico1	7	11
dinamico1	7	14
dinamico1	11	14

Conforme esperado, os três tipos de conflitos definidos na seção 3.2 foram detectados. As regras de condição-ação definidas pelo predicado *acao/1* funcionou como esperado.

O agente foi inserido em um controlador OpenFlow *Floodlight* para validar seu funcionamento em uma rede OpenFlow. Foi utilizado o *mininet* para emular uma rede *OpenFlow* com dois switches *OpenFlow* e dois hosts, onde cada switch possuia um host ligado a si.

O controlador foi modificado para operar no modo *proativo*, onde os fluxos devem ser adicionados previamente, caso contrário os elementos de rede não comunicarão entre si.

Foram inseridos fluxos sem conflitos e fluxos conflitantes (variações dos conflitos descritos na seção 3.2). Conforme esperado todos os conflitos, dinâmicos e estáticos, foram detectados.

Para detectar outros tipos de conflitos não definidos pelo presente trabalho, basta adicionar uma nova regra de condição-ação do predicado *acao/1* com o mesmo esquema proposto neste trabalho.

Foi realizado algumas avaliações de desempenho, visando mensurar o tempo necessário para resolver conflitos usando diferentes bases de conhecimento, conforme a Tabela 1.

Tabela 2 – Tempo para análise da tabela de fluxos

Quantidade de Fluxos	**Tempo**
10	0,001s
100	0,002s
1.000	0,010s
10.000	0,172s
20.000	0,311s
30.000	2,180s
40.000	5,177s
50.000	9,079s
60.000	17,993s
70.000	39,764s
80.000	71,116s
90.000	120,692s
100.000	257,675s

A Figura 5 sintetiza os dados da tabela 1. Como pode-se contatar, de forma empírica, o agente proposto possui ordem de complexidade $\Omega(n^2)$.

Figura 5 – Gráfico do tempo dispendido pelo agente ao processar fluxos

Comparando com o trabalho de Natarajan, Huang e Wolf (NATARAJAM; HUANG; WOLF, 2012) que conseguiu processar 3.600 fluxos entre 0,060 e 0,080 ms, o agente lógico proposto nesse trabalho processa 4.000 fluxos com 0,090 ms. Vale ressaltar que as tabelas de fluxos utilizadas são diferentes, pois o trabalho de (NATARAJAM; HUANG; WOLF, 2012) não descreve como foi montada a tabela de fluxos.

Com isso o tempo de resposta do algoritmo pode ser considerado boa, se considerarmos que um *switch OpenFlow* de 48 porta *Top-of-Rack* (*ToR*)trabalha com cerca de 5.000 fluxos por segundo.

Apesar de atender o caso médio perfeitamente (tempo de resposta menor que 0,172s) faz-se melhorar o tempo de resposta do agente para casos que poderá ser submetido a cargas maiores que 5.000 fluxos por segundo.

Outro detalhe a ser pontuado, é que o agente apenas entrará em ação toda vez que a tabela de fluxos for alterada pelo controlador *OpenFlow*, onde o mesmo irá avaliar a tabela de fluxos do switch, juntamente com os fluxos a serem inseridos, a existência de fluxos conflitantes.

Então, está não será uma operação recorrente e não ocasionará nenhum retardo na rede *OpenFlow* em questão.

6 Conclusão e Trabalhos Futuros

Conforme esperado, o mecanismo de detecção de conflitos implementado na forma de um agente lógico, conforme a síntese das arquiteturas proposta por Wooldridge (WOOLDRIDGE, 2001) e em Russel&Norving (RUSSELL; NORVIG, 2010) funcionou de forma racional, ou seja, alcançou seu objetivo comportando-se como esperado.

O agente lógico foi submetido a uma bateria testes em um controlador *OpenFlow*, validando seu funcionamento em um ambiente emulado, utilizando-se o *mininet* como ferramenta de emulação. Dessa forma é possível utilizar o agente lógico implementado nesse trabalho em um controlador *OpenFlow* que opere em uma rede real.

Contudo, observou-se alguns detalhes que, antes da implementação e dos testes, não estavam evidentes. Ficou evidente que, qualquer fluxo que siga as regras de condição ação definidas no agente lógico, será validado com sucesso, mesmo que o fluxo encontre-se fora do contexto da rede.

Para exemplificar, imagine que está se trabalhando com o mecanismo de detecção de conflitos proposto por esse trabalho em uma rede *OpenFlow* cuja a rede está definida, por exemplo, em 10.11.12.0/24. É possível adicionar um fluxo conflitante no controlador *OpenFlow* do tipo endereço IP de origem e endereço IP de destino iguais, por exemplo, mas o endereço IP é 172.24.5.33, por exemplo,

O agente lógico irá detectar o conflito, mas o endereço IP usado no fluxo conflitante está fora de contexto do endereçamento utilizado na rede *OpenFlow* supra citada. O controlador *OpenFlow* deve possuir alguma "inteligência"para poder revolver esse problema, podendo agir em conjunto com o agente lógico para alcançar esse objetivo.

Também observou-se a possibilidade de adicionar conflitos no agente utilizando informações extra protocolo *OpenFlow*.

Por exemplo, imagine que, em uma rede *OpenFlow*, é fornecido o serviço de voz sobre IP (VoIP) que utilize a porta UDP 27500 para transmissão de dados. Para estabelecer a comunicação entre dois hosts que desejam utilizar o serviço VoIP, necessita-se, no mínimo, de 300kbps de largura de banda. O agente lógico poderia verificar, via funcionalidade fornecida pelo controlador, a existência de largura de banda disponível antes de permitir que o controlador *OpenFLow* adicionar o fluxo e, caso não exista largura de banda suficiente, é notificado um conflito.

Outro ponto a ser destacado é a utilização dos campos do protocolo *OpenFlow* pelo agente lógico. A versão do protocolo *OpenFlow* utilizada nesse trabalho é 1.0. Dessa forma, atualmente, o agente não é portável entre versões do protocolo *OpenFlow*.

Uma abstração, em relação ao protocolo *OpenFlow*, poderia ser utilizada para

esconder os detalhes da versão do protocolo utilizada, onde cada versão seria tratada como um dialeto pelo agente. Uma linguagem de políticas, como por exemplo o PonderFlow (BATISTA; FERNANDEZ, 2014), poderia ser utilizada para especificar os fluxos *OpenFlow* e pelo agente lógico para detecção de conflitos.

Diante dos resultados do Capítulo 5 e do que fora discursado acima, como trabalhos futuros é sugerido:

- A criação de uma heurística que melhore o tempo de execução do agente lógico;
- Implementar um mecanismo dentro do controlador *OpenFlow* que utilize o raciocínio lógico do agente para detectar a adição de fluxos *OpenFlow* fora do contexto atual da rede;
- Adicionar o suporte a políticas à definição de fluxos *OpenFlow* que abstraia a versão utilizada pelo protocolo *OpenFlow* e permita que o agente lógico seja utilizado independentemente da versão utilizada, além de permitir a adição de novas versões do protocolo sem alterar as atualmente suportadas.

REFERÊNCIAS

AL-SHAER, E.; AL-HAJ, S. Flowchecker: configuration analysis and verification of federated openflow infrastructures. In: *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. New York, NY, USA: ACM, 2010. (SafeConfig '10), p. 37–44. ISBN 978-1-4503-0093-3. Disponível em: <http://doi.acm.org/10.1145/1866898.1866905>. Citado 3 vezes nas páginas 28, 30 e 41.

BATISTA, B. L. A.; CAMPOS, G. A. L. de; FERNANDEZ, M. P. A proposal of policy based OpenFlow network management. In: *20th International Conference on Telecommunications (ICT 2013)*. Casablanca, Morocco: [s.n.], 2013. Citado na página 31.

BATISTA, B. L. A.; CAMPOS, G. A. L. de; FERNANDEZ, M. P. Flow-based conflict detection in OpenFlow networks using First-Order logic. In: *19th IEEE Symposium on Computers and Communications (IEEE ISCC 2014)*. Madeira, Portugal: [s.n.], 2014. Citado na página 31.

BATISTA, B. L. A.; FERNANDEZ, M. P. Ponderflow: A policy specification language for openflow networks. *ICN 2014 : The Thirteenth International Conference on Networks*, p. 204–209, 2014. Citado 2 vezes nas páginas 31 e 56.

BRADEN, R. et al. Developing a next-generation internet architecture. *MIT Technical Report*, MIT, Massachusetts, USA, jul. 2000. Disponível em: <http://groups.csail.mit.edu/ana/Publications/DevelopingaNextGenerationInternetArchitecture.pdf>. Citado 2 vezes nas páginas 25 e 27.

CLARK, D. The design philosophy of the darpa internet protocols. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 18, n. 4, p. 106–114, ago. 1988. ISSN

0146-4833. Disponível em: <http://doi.acm.org/10.1145/52325.52336>. Citado na página 26.

CLARK, D. D. et al. Tussle in cyberspace: Defining tomorrow's internet. *IEEE/ACM Trans. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 13, n. 3, p. 462–475, jun. 2005. ISSN 1063-6692. Disponível em: <http://dx.doi.org/10.1109/TNET.2005.850224>. Citado 3 vezes nas páginas 25, 26 e 27.

COVINGTON, M. A.; NUTE, D.; VELLINO, A. *Prolog Programming in Depth*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN 0-13-138645-X. Citado 2 vezes nas páginas 37 e 38.

ERICKSON, D. *Floodlight Java based OpenFlow Controller*. 2014. Disponível em: <http://floodlight.openflowhub.org/>. Acesso em: 05.02.2014. Citado na página 33.

FELDMANN, A. Internet clean-slate design: What and why? *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 37, n. 3, p. 59–64, jul. 2007. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1273445.1273453>. Citado 3 vezes nas páginas 25, 26 e 27.

FERNANDEZ, M. *HermesNet: Framework para configuração e gerenciamento de redes OpenFlow*. 2014. Disponível em: <http://marcial.larces.uece.br/pesquisa/hermesnet>. Acesso em: 05.02.2014. Citado na página 33.

IBGE. *Censo Demográfico de 2010: Primeiros Resultados*. 2010. Disponível em: <http://www.ibge.gov.br/home/presidencia/noticias/imprensa/ppts/0000000237.pdf>. Acesso em: 15.01.2014. Citado na página 25.

IBGE. *Pesquisa Nacional por Amostra de Domicílios - 2012*. 2012. Disponível em: <ftp://ftp.ibge.gov.br/Trabalho_e_Rendimento/Pesquisa_Nacional_por_Amostra_de_Domicilios_anual/2011/tabelas_pdf/sintese_ind_8_1.pdf>. Citado na página 25.

LANTZ, B.; HELLER, B. *Mininet: rapid prototyping for Software Defined Networks*. 2012. Disponível em: <http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet>. Acesso em: 15.01.2014. Citado na página 29.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/1355734.1355746>. Citado 4 vezes nas páginas 27, 28, 33 e 34.

MIRZAEI, S. et al. Using alloy to formally model and reason about an openflow network switch. 2013. Citado na página 29.

MOURA, F. L. C. d. *Capítulo 10: A Lógica de Primeira Ordem*. 2014. Disponível em: <http://www.cic.unb.br/~flavio/disciplinas/graduacao/lc1/GramaticaLPredicados.pdf>. Citado na página 36.

NATARAJAM, S.; HUANG, X.; WOLF, T. Efficient conflict detection in flow-based virtualized networks. *International Conference on Computing, Networking and Communications*, p. 690–696, 2012. Citado 3 vezes nas páginas 28, 30 e 52.

NEC Corporation. *Trema Openflow Controller*. 2014. Disponível em: <http://trema.github.com/trema/>. Acesso em: 05.02.2014. Citado na página 33.

NOXRepo.org. *POX Openflow Controller*. 2014. Disponível em: <http://www.noxrepo.org/pox/about-pox/>. Acesso em: 05.02.2014. Citado na página 33.

ONF. *Software-Defined Networking (SDN) Definition*. 2014. Disponível em: <https://www.opennetworking.org/sdn-resources/sdn-definition>. Acesso em: 02.02.2014. Citado na página 27.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2010. (Prentice Hall Series in Artificial Intelligence). ISBN 9780136042594. Disponível em: <http://books.google.com.br/books?id=8jZBksh-bUMC>. Citado 6 vezes nas páginas 29, 31, 38, 46, 50 e 55.

SHERWOOD, R. et al. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, 2009. Citado na página 30.

SLOMAN, M. Policy driven management for distributed systems. *Journal of Network and Systems Management*, Vol.2, n. No 4, 1994. Citado na página 31.

STANFORD, U. *Clean Slate: A interdisciplinary Research Program*. 2014. Disponível em: <http://cleanslate.stanford.edu/>. Acesso em: 02.02.2014. Citado na página 27.

WOOLDRIDGE, M. J. *Introduction to Multiagent Systems*. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN 047149691X. Citado 4 vezes nas páginas 31, 46, 50 e 55.

Apêndices

A – Código Fonte do Agente Detector

O código fonte *Prolog* do Agente Detector é apresentado nesse apêndice. Lembramos que o código fonte está licenciado pela *GNU Public Licence v3*, conforme <http://www.gnu.org/licenses/gpl-3.0.txt>.

```
% Copyright 2014  Laboratório de Redes de Comunicação e Segurança - LARCES
%
% Este programa é um software livre; você pode redistribuí-lo e/ou
% modificá-lo dentro dos termos da Licença Pública Geral GNU como
%   publicada pela Fundação do Software Livre (FSF); na versão 2 da
%   Licença, ou (na sua opinião) qualquer versão.
%
%   Este programa é distribuído na esperança de que possa ser útil,
%   mas SEM NENHUMA GARANTIA; sem uma garantia implícita de ADEQUAÇÃO a qualquer
%   MERCADO ou APLICAÇÃO EM PARTICULAR. Veja a
%   Licença Pública Geral GNU para maiores detalhes.
%
%   Você deve ter recebido uma cópia da Licença Pública Geral GNU
%   junto com este programa, se não, escreva para a Fundação do Software
%   Livre(FSF) Inc., 51 Franklin St, Fifth Floor, Boston, MA  02110-1301  USA
%
%
% Agente Detector é formado por um sensor, um programa agente (detetor)
% e um atuador. Para realizar uma consulta:
%
% ?- detector.
%


detector:-
sensor('Repositorio de Politicas.txt',P),
detetor(P,A),
atuador(A,'Mensagens.txt').

/***********************************************/
% ?- sensor('Repositorio de Politicas.txt',P), write(P).
%
```

```
% O predicado sensor/2 abre o arquivo 'Repositorio de Politicas.txt'
% para leitura das políticas (que são predicados policy/6 que serão
% inseridos na base de conhecimento do agente Detector) e armazena o
% resultado em uma lista de políticas P, ou seja, representando as
% informações perceptivas de Detector, que são alimentadas para o
% programa agente detetor/2. Abaixo as informações perceptivas para o
% caso do arquivo em questão:

/*
 P = [policy(1, authmais, allowAddFlow, user, sw1, addFlow(1, _G8273,
 00:0e:2e:45:3c:79, 00:0e:2e:45:3c:79, _G8276, _G8277, _G8278, _G8279,
 _G8280, 192.168.0.1, 192.168.0.2, _G8283, _G8284, _G8285, _G8286,
 output:2)), policy(2, authmais, allowAddFlow, user, sw1, addFlow(1,
 _G8303, 00:0e:2e:45:3c:79, 08:54:77:ab:83:9a, _G8306, _G8307, _G8308,
 _G8309, _G8310, 192.168.0.1, 192.168.0.1, _G8313, _G8314, _G8315,
 _G8316, output:2)), policy(3, authmais, allowAddFlow, user, sw1,
 addFlow(1, _G8336, 00:0e:2e:45:3c:79, 08:54:77:ab:83:9a, _G8339,
 _G8340, _G8341, _G8342, _G8343, 192.168.0.1, 192.168.0.2, _G8346,
 _G8347, _G8348, _G8349, output:1)), policy(4, authmenos,
 denyAllPackets, user, sw1, addFlow(1, _G8372, 00:0e:2e:45:3c:79,
 08:54:77:ab:83:9a, _G8375, _G8376, _G8377, _G8378, _G8379, 11, _G8381,
 _G8382, _G8383, _G8384, _G8385, output:2)), policy(5, authmenos,
 denyAllPackets, user, sw1, addFlow(1, _G8408, 00:0e:2e:45:3c:79,
 _G8410, _G8411, _G8412, _G8413, _G8414, _G8415, 11, _G8417, _G8418,
 _G8419, _G8420, _G8421, drop))]
*/

sensor(Arq,P):-
see(Arq),
read(Política),!,
ler(Política,[],P),
seen.
sensor(_,[]):-
seen.

ler(P,L,L):-
P = end_of_file,!.
ler(P,Laux,L2):-
concatena(Laux,[P],Lauxnovo),
```

```
read(OutroP),
ler(OutroP,Lauxnovo,L2).

/************************************************/
% ?- sensor('Repositorio de Politicas.txt',P),detetor(P,A),nl,write(A).
%
% o predicado detetor/2 realiza o mapeamento P->A. Na realidade é uma
% concretização da arquitetura abstrata do agente com estado interno
% descrito por Wooldridge (ver-próximo-ação) e do programa agente
% simple reflex model based do Russell&Norvig do Russell (regras
% condição-ação). Para para as informações perceptivas acima ele gerou a
% seguinte ação, para ser alimentada para o atuador:
%
% A = [conflito(1, 1), conflito(2, 2), conflito(3, 3), conflito(4, [4, 5])].

detetor(P,A):-
ver(P,S),
próximo(S),
ação(A).

/************************************************/
% ?- sensor('Repositorio de Politicas.txt',P),ver(P,S),nl,write(S).
%
% o predicado ver/2 não precisou ser implementado realmente, pois as
% políticas no do arquivo 'Respo... .txt' já estão em forma de
% predicados, isto é não ocorre tradução, por isso, S=P:
/*
 S = [policy(1, authmais, allowAddFlow, user, sw1, addFlow(1, _G8273,
 00:0e:2e:45:3c:79, 00:0e:2e:45:3c:79, _G8276, _G8277, _G8278, _G8279,
 _G8280, 192.168.0.1, 192.168.0.2, _G8283, _G8284, _G8285, _G8286,
 output:2)), policy(2, authmais, allowAddFlow, user, sw1, addFlow(1,
 _G8303, 00:0e:2e:45:3c:79, 08:54:77:ab:83:9a, _G8306, _G8307, _G8308,
 _G8309, _G8310, 192.168.0.1, 192.168.0.1, _G8313, _G8314, _G8315,
 _G8316, output:2)), policy(3, authmais, allowAddFlow, user, sw1,
 addFlow(1, _G8336, 00:0e:2e:45:3c:79, 08:54:77:ab:83:9a, _G8339,
 _G8340, _G8341, _G8342, _G8343, 192.168.0.1, 192.168.0.2, _G8346,
 _G8347, _G8348, _G8349, output:1)), policy(4, authmenos,
 denyAllPackets, user, sw1, addFlow(1, _G8372, 00:0e:2e:45:3c:79,
 08:54:77:ab:83:9a, _G8375, _G8376, _G8377, _G8378, _G8379, 11, _G8381,
```

```
 _G8382, _G8383, _G8384, _G8385, output:2)), policy(5, authmenos,
 denyAllPackets, user, sw1, addFlow(1, _G8408, 00:0e:2e:45:3c:79,
 _G8410, _G8411, _G8412, _G8413, _G8414, _G8415, 11, _G8417, _G8418,
 _G8419, _G8420, _G8421, drop))]
*/

ver(P,S):-
S=P.

/************************************************/
% o predicado policy/6 é um predicado que será inserido por próximo na
% base de conhecimento do agente, por isso, a necessidade de declará-lo
% como dynamic

:-dynamic policy/6.

% o predicado próximo/1 insere ma base de conhecimento do agente cada
% uma das políticas na lista de políticas S, ao final do processamento
% do Agente é importante limpar a base de conhecimento.

próximo([]):-!.
próximo([X|Y]):-
assertz(X),
próximo(Y).

/************************************************/
% conforme mencionei acima, o mecanismo de tomada de decisão do agente é
% baseado em regras condição-ação.

ação(A):- faça_todas(A).

/************************************************/
% foi necessário criar uma definição faça_todas/1 que fosse capaz de
% verificar todas as regras condição-ação do agente e retornar as ações
% daquelas regras com as condições ativadas, ou seja, em uma lista de
% ações A.

:- dynamic lista_mensagens/1.
```

```
faça_todas(_):-
asserta(lista_mensagens([])),
faça(M),
retract(lista_mensagens(LM)),
        concatena(LM,[M],LM1),
asserta(lista_mensagens(LM1)),
fail.
faça_todas(A):-
retract(lista_mensagens(A)),
abolish(policy/6).

/************************************************/
% estas são as regras condição-ação do agente, por enquanto, foi
% necessário uma regra para cada conflito

faça(conflito(1,N)):-
policy(N,_,_,_,_,addFlow(_,_,EtherSrc,EtherDst,_,_,_,_,_,_,_,_,_,_,_,_,_,_)),
EtherSrc==EtherDst.
faça(conflito(2,N)):-
policy(N,_,_,_,_,addFlow(_,_,_,_,_,_,_,_,_,_,IpSrc,IpDst,_,_,_,_,_)),
IpSrc==IpDst.
faça(conflito(3,N)):-
policy(N,_,_,_,_,addFlow(IngressPort,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,output:Port)),
IngressPort==Port.
faça(conflito(4,[N1,N2])):-
     policy(N1,authmenos,denyAllPackets,U,S,addFlow(IngressPortX,_,EtherSrcX,_,_,_,
     policy(N2,authmenos,denyAllPackets,U,S,addFlow(IngressPortY,_,EtherSrcY,_,_,_,
     IngressPortX==IngressPortY,
     EtherSrcX==EtherSrcY,
     IpSrcX==IpSrcY.

/************************************************/
% predicado atuador/2 escreve no arquivo 'Mensagens.txt' a lista de
% ações A, detectada por faça_todas/1, em um formato mais adequado, por
% exemplo:
%
% ?- atuador([conflito(1, 1), conflito(2, 2), conflito(3, 3),conflito(4, [4, 5])],'

atuador(A,Arq):-
```

```
tell(Arq),
escrever(A),
told.

escrever([]):-
!,write('Não existe conflito '),nl.
escrever([conflito(Tipo,N)]):-
!,write('Conflito tipo '),write(Tipo),
write(' na(s) política(s) '),write(N),nl,
mensagem(Tipo).
escrever([conflito(Tipo,N)|Y]):-
write('Conflito tipo '),write(Tipo),
write(' na(s) política(s) '),write(N),nl,
mensagem(Tipo),
escrever(Y).

mensagem(1):-
!,write('Campo etherSrc e etherDst iguais'),nl,nl.
mensagem(2):-
!,write('Campo ipSrc e ipDst iguais'),nl,nl.
mensagem(3):-
!,write('Saída do switch redirecionada para mesma porta em ingressPort'),nl,nl.
mensagem(4):-
write('Descarte do que entra em insgrePort do switch com o MAC em etherSrc').

/***********************************************/
concatena([],L,L):- !.
concatena([X|Y],Z,[X|YZ]):-
concatena(Y,Z,YZ).
```