UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DE DESEMPENHO COMPARATIVO ENTRE EMULAÇÃO, SIMULAÇÃO E MÉTODOS HÍBRIDOS PARA SDN

JEAN MENOSSI

ORIENTADOR: PROF. DR. CÉSAR A. CAVALHEIRO MARCONDES

São Carlos – SP

Janeiro/2017

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DE DESEMPENHO COMPARATIVO ENTRE EMULAÇÃO, SIMULAÇÃO E MÉTODOS HÍBRIDOS PARA SDN

JEAN MENOSSI

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 e Sistemas Distribuídos

Orientador: Prof. Dr. César A. Cavalheiro Marcondes

São Carlos – SP

Janeiro/2017

UNIVERSIDADE FEDERAL DE SÃO 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 Jean Menossi, realizada em 24/02/2017.

Prof. Dr. Cesar Augusto Cavalheiro Marcondes
(UFSCar)

Prof. Dr. Hermes Senger
(UFSCar)

****
Profa. Dra. Natalia Castro Fernandes
(UFF)

Certifico que a sessão de defesa foi realizada com a participação à distância do membro Natalia Castro Fernandes, depois das arguições e deliberações realizadas, o participante à distância está de acordo com o conteúdo do parecer da comissão examinadora redigido no relatório de defesa do aluno Jean Menossi.

Prof. Dr. Cesar Augusto Cavalheiro Marcondes
Presidente da Comissão Examinadora
(UFSCar)

”Aquele que vê todos os seres no Eu, e o Eu em todos os seres, não odeia ninguém”

Os Upanishads - Sopro do Eterno

RESUMO

Desde sua concepção Redes Definidas por Software teve um grande crescimento e adoção em diversas áreas, explorando e expandindo cada vez mais as formas de utilização dessa tecnologia. Para desenvolver uma aplicação utilizando o protocolo OpenFlow são necessários recursos que possuem suporte para tal, controlador e ambiente de testes sendo que estes podem ser emulados, simulados ou mesmo com a reserva de uma testbed. Este trabalho procura explorar as diferenças de implementação e o *tradeoff* entre esses ambientes bem conhecidos, apontando números que sejam interessantes na hora da escolha de qual método utilizar. Os ambientes se mostram equiparáveis em desempenho com links de baixa transmissão de dados como 100MB e 1GB, porém tem uma diferença significativa em relação a links de 10GB, além de diferenças de modelagem e uso particulares de cada um.

Palavras-chave: RDS, NS-3, Emulação, Simulação

ABSTRACT

Since its conception Software Definined Network had a great growth and adoption in several areas, exploring and expanding more and more forms to use this technology. For development of an application using OpenFlow protocol, some resources are needed as assets that support it, controller and test environments that can be used as emulated, simulated or even scheduling resources in a testbed. This work aims to explore the differences of implementation and tradeoff between these well-known environments and the trade, pointing numbers that are interesting to choose a method. The environments are comparable in performances with links of low data transmission like 100MB and 1GB however it has a significant difference in with links of 10GB, besides differences of modeling and particular use of each one.

Keywords: SDN, NS-3, Emulation, Simulation

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

Capítulo 1

INTRODUÇÃO E MOTIVAÇÃO

Dentre as diversas áreas disponíveis para pesquisa em computação, a área de Redes, desde sua concepção, tem sido estática, com apenas propostas de melhoramento de uma estrutura já existente, mas nunca com uma mudança total de paradigma. Observando esse engessamento e o, cada vez maior, acúmulo de protocolos e propostas de novos algoritmos, um grupo de pesquisadores de diversas universidades propõe um novo protocolo (OpenFlow(MCKEOWN, 2008)), o qual mudou o layout e a forma de pensamento e desenvolvimento em Redes. O OpenFlow tem como principal objetivo separar as camadas de dados e controle, criando um ponto central de decisão para os dispositivos. Essa forma de visualizar redes foi concebida com o nome de Redes Definidas por Software, ganhando cada vez mais destaque em pesquisa e indústria. A proposta foi, em vista de seu caráter prático e assertivo, tão bem recebida tanto por pesquisadores quanto por fabricantes que, em 2011, foi fundada a *Open Network Foundation*(ONF,), dedicada à regulamentação das Redes Definidas por Software e do protocolo OpenFlow de forma aberta à colaboração dos membros. Desde sua origem, diversos trabalhos envolvendo otimização, casos de uso e desenvolvimento de aplicações para SDN têm sido publicados, porém ambientes de testes para esse tipo de pesquisa são limitados e muitas vezes não são capazes de atender a todos os recursos que a aplicação desenvolvida se propõe a solucionar. Apesar de promissor, o protocolo OpenFlow em hardware ainda é caro, tornando-o inviável para algumas modalidades de pesquisa. A maioria das pesquisas desenvolvidas em Redes Definidas por Software fazem uso, majoritariamente, de mecanismos de simulação e emulação. A criação do ambiente Mininet(LANTZ; HELLER; MCKEOWN, 2010), por exemplo, revolucionou a forma de uso de *containers* para ensino e pesquisa. Tendo como base o OVS (OPEN...,) (outra tecnologia desenvolvida a partir da ideia que Redes Definidas por Software introduziu), os ambientes de redes podem ser criados, com interação real e verossímil, de uma forma bem simples. Essa ferramenta, conhecida por sua simplicidade de uso e programação (MININET..., b), tem diversos

trabalhos baseados em sua plataforma (AL-FARES, 2010)(LACAGE, 2010)(HUANG; YOCUM; SNOEREN, 2013)(NUNES, 2014), sempre voltados ao princípio de Redes Definidas por Software. Mas além de emulação, existe outro método? Sim! Ainda que este seja de fácil utilização, existem modos mais complexos de se lidar diretamente com emulação. Se a aplicação desenvolvida pretende utilizar diversos links com uma alta carga de dados, por exemplo, pode ser que a máquina hospedeira não seja suficiente para entregar os resultados sem ruídos. Nesse contexto, a simulação tem seu papel garantido. Diversos simuladores (NS...,)(TECHNOLOGIES,)(GUPTA; SOMMERS; BARFORD, 2013) foram desenvolvidos ou têm suporte para Redes Definidas por Software. Além disso, tanto em emulação quanto em simulação, existem diversas pesquisas que têm como base uma tecnologia para melhor entrega de resultados, métricas ou soluções de problemas antigos presentes na abordagem comum de Redes.

Em questões de segurança, por exemplo, existem artigos e trabalhos voltados diretamente para a aplicação em Redes Definidas por Software (SHIN, 2013b)(BARI, 2013), bem como a diversificação da arquitetura e modo de aplicação da tecnologia (JONDRAL, 2005)(JAIN, 2013)(LUO; TAN; QUEK, 2012)(JAIN; PAUL, 2013)(NASCIMENTO, 2011) e controladores que possam satisfazer tanto o usuário acadêmico, em sua pesquisa, quanto o ambiente de produção industrial, no caso de uma empresa que deseje adotar esse novo paradigma em sua rede(NOXREPO.ORG,)(KREUTZ, 2015)(SHIN, 2013a). Um exemplo claro de como a aplicação de Redes Definidas por Software tem sido amplamente aceita e adotada pelo meio industrial, pode ser observado pelo desenvolvimento de soluções e apadrinhamento do projeto feito por empresas como Google, HP, IBM e Telefônica (ONF,)(NUNES, 2014). A HP, recentemente, entregou um controlador modular que pode acumular diversas aplicações desenvolvidas para diferentes fins (HP-VAN...,), o que demonstra um compromisso na adoção e desenvolvimento da tecnologia. Outro exemplo interessante de ser citado é a plataforma Fibre (FIBRE,), uma *testbed* que tem recursos para testes em Redes Definidas por Software, essa plataforma que se tornaria o principal meio de pesquisa, em cima de um ambiente real, para as universidades brasileiras, demanda um grande investimento ser feito, atestando sua relevância. Mas o que motiva essa pesquisa, então? Se existem diversas formas de aplicar os conhecimentos de desenvolvimento em Redes Definidas por Software? Entra, aqui, a principal motivação: mesmo com tantas plataformas, cada uma tem sua limitação de uso. Portanto, é necessário entender os limites de uso e dificuldade de desenvolvimento antes mesmo de criar uma solução em Redes Definidas por Software, afinal, é difícil pensar em uma pesquisa séria sem testes e resultados que possam comprovar a eficácia pretendida.

Uma comparação da dificuldade do desenvolvimento entre as plataformas, além do desempenho adquirido em cada uma a partir de uma aplicação, pode determinar qual deve ser utilizada

e se existem ganhos ou perdas na troca. Este tipo de pesquisa aponta diretamente, ao experimentador ou comunidade, quais os melhores recursos a serem utilizados para um determinado fim de pesquisa, obtendo assim resultados mais próximos possíveis de um ambiente real. Ambientes diferentes demandam resultados e um desenvolvimento diferenciado. Um *layout datacenter* não exige o mesmo tipo de recurso de um *layout* em estrela ou *wireless*. Vistas essas necessidades, tanto de arquitetura quanto de entrega final dos recursos, qual a melhor forma de testes? Essa é a pergunta que embasa a pesquisa deste trabalho.

1.0.1 Objetivos do Trabalho

Este trabalho objetiva entregar uma comparação de desempenho entre o uso de simulação, emulação e métodos híbridos para o desenvolvimento de aplicações em Redes Definidas por Software, como resposta prática para se lidar com os limites de implementação de cada tecnologia. Também visa levantar dados suficientes para apontar novas soluções de lapsos para simuladores, emuladores, métodos híbridos e *testbeds*.

1.0.2 Organização do Trabalho

Este trabalho está subdividido em quatro capítulos principais, sendo iniciado pela Fundamentação Teórica, que prevê cobrir todo conhecimento necessário para melhor compreensão de toda interpretação dos dados presentes em Implementação e Resultados. A própria Implementação, a qual detalhará todo esforço técnico desenvolvido, juntamente com os resultados obtidos em cada sessão e bateria de testes - apresentando, assim, a sumarização deste trabalho; e, por fim, a Conclusão, que se propõe a apresentar uma discussão dos resultados e propostas para trabalhos futuros a serem desenvolvidos a partir dos dados obtidos.

Capítulo 2

FUNDAMENTAÇÃO TEÓRICA

Este capítulo compreende as tecnologias envolvidas e os conceitos empregados para o entendimento da proposta e seu desenvolvimento.

2.1 OpenFlow e Redes Definidas por Software

O protocolo OpenFlow, hoje em sua versão 1.3 (PFAFF, 2012), possibilita um controle refinado da rede e de recursos nunca antes alcançados pelas redes (KREUTZ, 2015),(NUNES, 2014),(HU; HAO; BAO, 2014). Essa sessão pretende descrever o protocolo 1.0, especificação e documentação *Open Network Foundation* (ONF,).

No documento de especificações do protocolo, desde sua versão 1.0 (CONSORTIUM et al., 2009) , são definidos os requisitos que um hardware necessita para ter o protocolo em pleno funcionamento como parte de um *switch*, seja físico ou virtual, utilizando ou não *Network Function Virtualization*. Um *switch* Openflow é composto por uma *flow table*, que faz a pesquisa de pacotes e repasse, portas ativas para OpenFlow e um canal seguro de comunicação entre o switch e o controlador Openflow 2.1 (CONSORTIUM et al., 2009). Todos os pacotes recebidos pelo *switch* OpenFlow são comparados com a *flow table*. Se existir uma combinação de pacote com alguma entrada da tabela, a ação especificada é tomada, caso não exista nenhuma entrada correspondente, o pacote é então repassado para o controlador pelo canal seguro. Outras ações são possíveis, dependendo da versão do protocolo, as quais serão abordadas nas próximas seções.

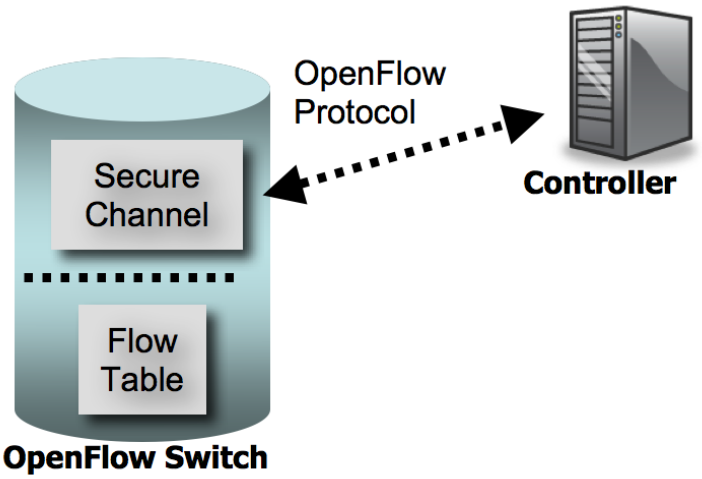

Figura 2.1: Comunicação OpenFlow.

2.1.1 OpenFlow 1.0

Essa versão foi a primeira oficial com suporte ao vendedor, versões anteriores eram consideradas rascunhos e, portanto, não era recomendada a construção de nenhum *switch* a partir dessas especificações. Partindo da estrutura básica já apresentada, o *switch* com suporte a OpenFlow é composto pela *flow table*, portas OpenFlow e um canal seguro para comunicação entre o *switch* e o controlador OpenFlow, que ainda pode ser dividido em duas classes: *Switch* puramente OpenFlow e híbrido, este último permite as funções comuns de um *switch* comum, e o OpenFlow, que pode ser ativado apenas nas portas necessárias. As portas OpenFlow nesta versão são limitadas aos estados de ligada, desligada e uma função *Spanning Tree*, a qual determina por onde os pacotes deveriam sair. As especificações desta versão colocam a *flow table* formada pelos seguintes campos demonstrados na Figura 2.2:

Figura 2.2: *FlowTable* OpenFlow 1.0.

Em que o cabeçalho são as informações do pacote que devem ser inspecionadas, o contador um campo que atualiza a quantidade de pacotes que combinaram com o cabeçalho do campo anterior e as ações que determinam o que deve ser feito com aquele determinado fluxo.

Cada desenvolvedor é livre para escolher qual ou quais campos devem combinar para uma determinada ação. Dada a combinação(*match*), o contador daquela tupla específica é incrementado e a ação é realizada. Se a ação for zero, ou seja, nenhuma, o pacote é descartado. Como as ações podem ser várias para um único fluxo ou pacote, é possível implementar uma lista

de ações que são executadas de forma sequencial àquela em que foram inseridas. Todo *switch* OpenFlow deve conter obrigatoriamente as seguintes ações em portas físicas:

- *ALL*: Envia o pacote para todas as interfaces no *switch*
- *CONTROLLER*: Encapsula e envia o pacote para o controlador.
- *LOCAL*: Envia o pacote para a pilha local do *switch*.
- *TABLE*: Executa ações no tabela de fluxos.
- IN PORT: Envia o pacote para a porta em que entrou.

Existem ainda ações opcionais (que podem ou não ser implementados em um *switch* OpenFlow) que não serão especificadas. São ações de portas virtuais para um *switch* NFV, por exemplo. Compõem, portanto: *Normal, Flood, Modify-Field*, os quais adicionam os recursos de VLAN, *Flood* STP e modificação de um campo específico no pacote (Destino IPv4, porta de origem ou destino - dentre outros), respectivamente. Uma ação para todos os tipos de *switch* é a ação de *DROP*, para simplesmente descartar um pacote caso exista uma combinação. Um bom exemplo para esta ação de *DROP* é uma aplicação *firewall*, descartando pacotes de uma porta ou serviço específico.

O processo de combinação segue um fluxo único, mostrado na figura 2.3:

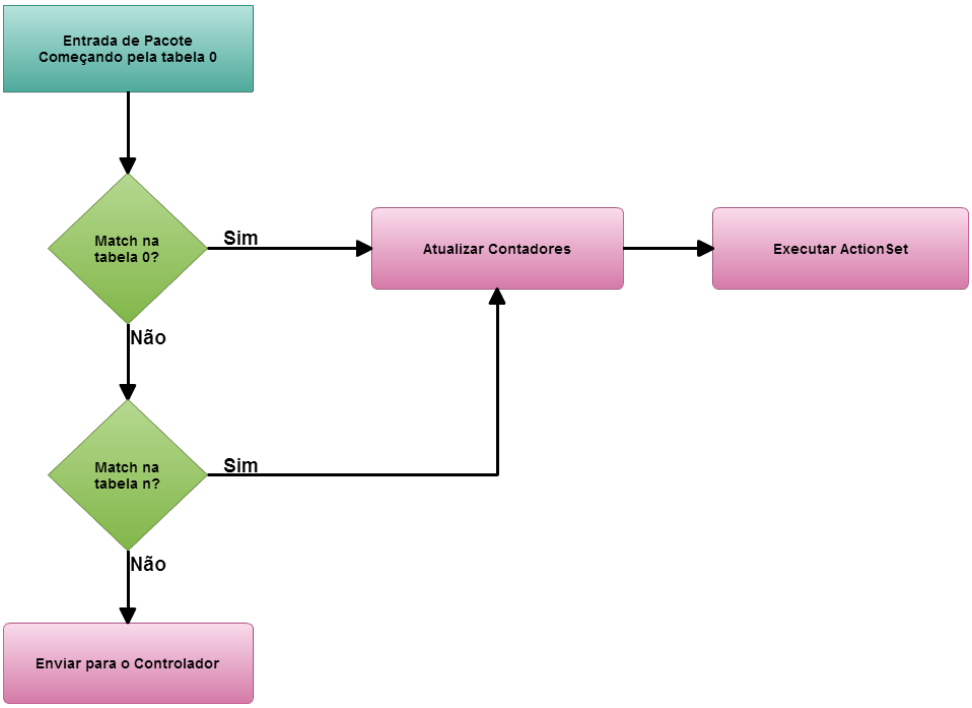

Figura 2.3: Fluxograma de Match.

Seguindo o fluxograma da tabela figura 2.3, quando um pacote chega da rede (*packet in)*, o *switch* verifica na *flow table* se existe algum campo do cabeçalho do pacote que tenha combinação com alguma regra instalada na *flow table*. Caso haja uma combinação, as ações configuradas para este fluxo de pacotes serão tomadas, em ordem sequencial. Isso quer dizer que podem haver mais ações em uma combinação na *flow table*, atribuindo então ao campo uma lista de ações instaladas, que são seguidas sequencialmente. Um cenário prático seria a cópia do pacote para um servidor de análise IDS (*Intrusion Detection System*) antes de tomar uma ação de repasse ou descarte, por exemplo. Todas as tabelas são analisadas sequencialmente até que em alguma delas (não é necessário que exista mais de uma tabela, mas pelo menos uma) haja uma combinação. Se todas as tabelas forem percorridas e nenhuma combinação for encontrada, então o pacote é enviado para o controlador pelo canal seguro para que ele tome a decisão do que deve ser feito, respondendo com uma regra para aquele fluxo de pacotes. Uma opção é determinar que o *switch* descarte o pacote, caso nenhuma tabela possua uma combinação em nenhum dos campos do cabeçalho do pacote. Alguns pontos devem ser considerados: se estiver configurado para *ANY*, qualquer campo do pacote pode servir como uma combinação; uma combinação direta em alguma tupla da *flow table* sempre terá maior prioridade na execução das ações; para cada pacote em que uma combinação existir, o contador é incrementado e, nesta versão, não existe suporte para múltiplos controladores.

Com essa estrutura de funcionamento, fica mais fácil entender de uma forma técnica como a imagem 2.3 funciona. Para se estabelecer uma conexão, o *switch* tenta se comunicar com o controlador em um IP pré-configurado pela porta padrão 6633 (alterado para 6653 (DENAZIS, 2015)). Nesta tentativa de conexão, cada um dos lados envia uma mensagem do tipo *Symetric* (tipo de mensagem que não necessita de solicitação), com um OFPT HELLO contendo a versão mais alta de OpenFlow suportada. Com isso, ambos os lados calculam qual a menor versão OpenFlow disponível que será utilizada se ambos os lados tiverem uma versão compatível. Então, a conexão é inicializada. Caso não exista nenhuma versão compatível ou ocorra algum erro durante essa troca de mensagens inicial, o lado (seja controlador ou *switch*) deve enviar uma mensagem de OFPT ERROR com o campo OPPET HELLO FAILED. Nesta versão do protocolo não existe nenhuma forma de se utilizar múltiplos controladores com suporte direto (que o próprio protocolo OpenFlow trate). Ainda se tratando de conexão entre o *switch* OpenFlow e o controlador, é necessário pensar o procedimento caso ocorra um rompimento de comunicação entre os pontos, seja por um ”*echo request*”sem reposta ou qualquer outro *timeout*. Para tal, o protocolo provê um ”*emergency mode*”que continua tentando restabelecer a conexão com o controlador, reiniciando a conexão TCP e exclui todas as regras da *flow table* que não estejam setadas com a *flag* de *emergency* ativa. Um detalhe de implementação, é abstrair quais fluxos

têm grande importância e lembrar de setá-los com o modo de *emergency* ativo. A comunicação com o controlador usa um canal criptografado por TLS.

Com a conexão estabelecida, os fluxos começam a ser passados para o controlador decidir qual ação executará sobre eles. Essa troca de mensagens também é definida pelo protocolo OpenFlow: como as regras serão instaladas e modificadas dentro da *flow table*. Essas mensagens são divididas em *ADD, MODIFY, MODIFY STRICT, DELETE* e, por fim, *DELETE STRICT*, sendo que cada uma tem suas regras de decisão em caso de sobreposição ou conflito na tabela de fluxos. Quando um *switch* recebe uma requisição para adicionar uma regra (*ADD*) com a *flag "CHECK OVERLAP"* ativada, o *switch* procura pela mesma regra na *flow table*, com a mesma prioridade. Se for encontrada uma regra exatamente igual, o *switch* recusa a regra, retornando uma mensagem de erro. Caso a regra não tenha essa *flag "CHECK OVERLAP"* ativada, a regra é simplesmente instalada. Funciona da mesma forma para regras novas (que ainda não existam na tabela), sendo estas inseridas com o menor número da tabela.

Ainda nesse processo de adição, já foi abordado que a tabela utiliza de recurso físico do *switch*, tornando-se limitada. Portanto, é possível que a tabela não tenha espaço suficiente para novos fluxos. Neste caso, retorna ao controlador uma mensagem de erro (OFPT ERROR) apontando que todas as tabelas estão cheias e nenhum novo fluxo pode ser inserido. Esse erro determina a importância de se abstrair corretamente o fluxo da rede e pensar nas medidas de *overlap* possíveis para serem implementadas. Ainda, caso a regra utilizar de uma porta que não existe ou não está alocada para o protocolo OpenFlow (no caso de switches híbridos ou mesmo virtuais), uma mensagem de erro também retorna apontando esse problema. Mesmo se a porta for adicionada no chassi do *switch* ou virtualmente, ela irá descartar os pacotes até que seja devidamente referenciada.

Em sua versão 1.0, o OpenFlow fornece um mecanismo simples de fila para prover a função de qualidade na qual, uma fila associada a uma porta pode mapear os fluxos que entram por ela. Fluxos mapeados por uma fila serão tratados pelas configurações que a própria fila impõe, como exemplo uma taxa mínima de transmissão. Para que esse recurso funcione, é necessário enviar o fluxo para a fila, que necessita da ação de *Enqueue*, tratada como opcional nas especificações.

Portanto, foi abordado todo funcionamento básico e alguma extensão de componentes opcionais a um *switch* OpenFlow em sua versão 1.0.

2.2 Testbeds, Simuladores e Emuladores

As alternativas presentes para um pesquisador ou estudante de redes desenvolver suas aplicações ou executar seus experimentos concentram-se em três frentes mais conhecidas: *testbeds*, simuladores e emuladores. As seções a seguir vão discutir brevemente cada uma das tecnologias, com, ao final, uma breve reflexão comparativa entre os pontos apresentados.

2.2.1 Testbeds

Criar um ambiente de testes de campo, além de ter um alto custo, pode apresentar grande complexidade de configuração e demandar muito tempo para um *layout* simples. *Testbeds* proveem para seus usuários uma forma mais simples de ter um ambiente real para testes, podendo adequar até certo ponto as necessidades específicas do usuário, provendo recursos de hardware, links de comunicação, máquinas virtuais e etc. Mesmo com acesso a tantos recursos físicos, uma *testbed* pode ser um problema, visto que seus recursos são limitados no sentido de sua organização física e disponibilização para o experimentador. Nesse contexto de uso do protocolo OpenFlow, a *testbed* precisa fornecer *switches* com suporte ao protocolo, máquinas virtuais, conexões diversificadas e se possível links federados (com comunicação entre várias outros pontos).

Um exemplo próximo para o experimentador brasileiro é o projeto Fibre (FIBRE,), que conta com diversas universidades e além de uma ampla infraestrutura física, interconectando diversos recursos para experimentação. O projeto conta com ”ilhas”, representadas e mantidas por cada uma das universidades participantes, que tem parte da infraestrutura comum e alguns recursos específicos de acordo com a linha de pesquisa ou interesse da instituição:

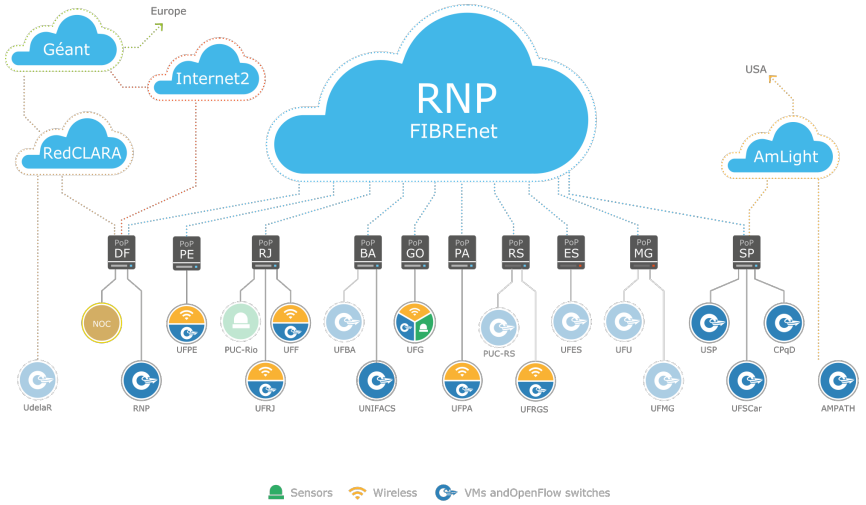

Figura 2.4: Rede e recursos por ilha participante do projeto (FIBRE,)

A figura 2.4 demonstra como ocorre a interligação entre as instituições, bem como os recursos específicos de cada uma. É possível notar que existem basicamente três tipos de estruturas disponíveis:

- VM e OpenFlow: Disponibiliza no rack, infraestrutura de virtualização para máquinas virtuais e recursos OpenFlow;
- Wireless: Nós de teste sem fio para pesquisa, possuem diversas antenas e são reservados para uso exclusivo, sem a necessidade de partilha do hardware;
- Sensores: iniciativa de uma rede de sensores ainda em desenvolvimento e implementação.

Todas as ligações representadas na figura 2.4 são reservas em uma topologia física composta por fibra óptica. Assim sendo, qualquer ilha tem uma garantia de entrega (não dependendo apenas do layout apresentado, mas da implementação da instituição ou universidade) com qualidade e pouco delay. Todas as ilhas possuem um rack de servidores com os seguintes equipamentos:

Tabela 2.1: Recursos por ilha Fibre (FIBRE,)

Amount	Hardware	Purpouse
1	IBM Server	Virtualization server
2	Dell Server	PerfSONAR monitoring tool
3	NetFPGA Server	OpenFlow resource
1	Datacom DM 4100	Top of Rack
1	Pronto PICA8	OpenFlow resource
∼3	ICARUS	Wireless tests

Com os recursos acima apresentados (que podem variar de ilha para ilha) e a topologia que os interligam, é possível recriar diversas topologias sobrepostas para experimentação. Nesse contexto é importante elucidar que, uma *testbed* nesse patamar de cobertura de rede e virtualização, necessita de uma forma de controle refinada. Toda a infraestrutura, desde o cadastro dos experimentadores como o agendamento de recursos é controlado pelo *Ofelia Control Framework* (OCF)(FP7,). Seus módulos foram preparados para que possua virtualização com o OXA (*Ofelia Xen Administrator*) e utilização do *Flowvisor* (SHERWOOD, 2009), para permitir múltiplos experimentos OpenFlow em um único recurso de hardware compartilhado. O Flowvisor atua como um *proxy* entre a topologia requerida pelo experimentador e a topologia real, apresentando para o controlador OpenFlow apenas o que foi requerido. O que fica claro nessa forma de utilização é um ponto que interligue as redes do usuário para o controlador e a topologia física, adicionando um *overhead* para o funcionamento e isolamento das redes, que pode alterar parte dos resultados.

Os recursos *wireless* são de uso exclusivo, ou seja, é feita a reserva dos nós e eles não tem seu recurso compartilhado. O controle dos nós e a experimentação que necessita ser feita gira em torno do OMF. Toda a *testbed* pretende migrar para o mesmo, mas por hora mantém a estrutura já apresentada. A utilização de OMF prevê que o experimentador conheça linguagem e, ao rodar seu experimento criado em OMF todos os recursos sejam automatizados. Uma execução de pesquisa escrita na linguagem configura e executa comandos a partir do que for determinado, sendo assim toda a execução é automatizada, bem como o tratamento das configurações básicas e coleta dos resultados. A cada vez que o nó é reservado para receber um experimento, ele *wireless* é formatado e preparado com configurações padrão para a execução. Feita a coleta de dados, a próxima reserva de recursos recomeça o processo de configuração do nó e preparo para o experimento a partir do que for requisitado, reservado e escrito na linguagem OMF.

Nesse contexto, um pesquisador pode ter seu ambiente preparado para o teste em específico, em equipamento real, com garantia de fidelidade. Porém, o acesso a esses recursos são mediante aprovação, por tempo limitado e compartilhado com outros usuários, o que pode impactar no resultado final.

2.2.2 Simuladores

A ideia do por trás do uso de simulações, vem do uso de algoritmos matemáticos que representem determinado objeto a ser simulado. Tendo esse contexto em mente, os pacotes e os eventos que ocorrem em uma simulação são apenas fictícios e as trocas de dados, bem como o resultado, partem apenas de cálculos representativos. O comportamento da rede portanto,

não deve alterar durante suas execuções, vantagem clara de ter resultados iguais ou similares a cada rodada de execução da simulação. Durante sua execução deve-se lembrar que não existem dados reais ou mesmo recursos que demandem da máquina hospedeira da simulação, todo acontecimento é representativo e não uma ação real dentro de um sistema, apesar de ser possível gerar dados reais a partir de uma simulação, visto que toda execução é baseada em tratamento e condições reais. Os algoritmos matemáticos devem representar de forma fidedigna o mundo que pretende simular. A vantagem clara do uso de algoritmos é possibilidade de utilizar recursos que não estão disponíveis, como uma execução do comportamento de 10 anos em uma rede ou links e equipamentos operando a 100 Gigabits por segundo.

2.2.3 Emuladores

Apesar das muitas características positivas na simulação, ela fornece apenas uma modelagem matemática. Pontos de consumo de memória, utilização de processamento compartilhado, gargalos de troca de mensagens ou mesmo problemas ocultos no sistema operacional não podem ser simulados, simplesmente por não serem previsíveis. A emulação permite o uso de *hardware* simples, com poucos recursos para representar uma grande rede. Neste contexto da emulação, os links, processos e aplicações consomem de acordo com a necessidade e podem entregar de diversas formas possíveis resultados, diferenciados conforme sua execução.

Emulação portanto utiliza dados e recursos reais para a execução de um *layout* e tem que compartilhar com todo nó emulado, além do mínimo requisitado para o sistema hospedeiro. Como os dados e as interações são reais, diversas ondulações podem aparecer nos resultados, visto que tudo depende da máquina hospedeira. Assim sendo, os links e o tempo de emulação estão fortemente relacionados a máquina. Utilizando o mesmo exemplo da seção anterior, observar o comportamento de uma rede durante 10 anos com o link de 100 Gigabits necessitaria de uma execução de 10 anos e uma máquina que comportasse durante esse tempo as transações entre os links a 100 Gigabits.

2.2.4 Comparação

Em uma comparação entre simuladores, emuladores e testbeds temos o seguinte resultado apresentado na tabela 2.2:

Tabela 2.2: Tabela de comparação entre ambientes de testes

Simuladores	Emuladores	Testbeds
-Modelo	-Modelo	+Sistema Real
-Simplificado	+Preciso	+Preciso
++Escalável	+Escalável	-Não Escalável
++Rápido	+Rápido	-Lento
++Flexível	+Flexível	-Não Flexível
++Pode ser repetido	++Pode ser repetido	-Não pode ser repetido
++Barato	++Barato	-Caro
++ Sem administrador	++Sem administrador	-Com administrador

Como pode ser observado, a vantagem do sistema real, como um espaço em uma *testbed*, é acompanhado por diversos fatores negativos dependendo da amplitude e profundidade que o pesquisador pretende explorar. Para pesquisas nas quais o ambiente será fixo, quando os recursos forem suficientes, a precisão e a facilidade de manuseio de uma *testbed* é uma solução barata e rápida, contudo, seus meios podem não variar durante o tempo, entregando sempre o mesmo *layout*. Os recursos compartilhados entre diversos usuários podem criar flutuações no resultado se este não for bem isolado pela tecnologia e, normalmente, existem *overheads* que influenciam no estudo dos dados finais, como o uso do *Flowvisor*, ferramenta intrínseca da *tesbed* Fibre (em pesquisas envolvendo OpenFlow). Outro ponto a ser discutido, mesmo que minimamente, é o custo fixo e tempo para alocação também necessárias para o mesmo fim. O pesquisador pode ter acesso à plataforma, mas não tem um controle refinado e detalhado dos links e da transição de dados, apesar do resultado ser totalmente fiel ao uso de uma plataforma real, existem pontos a ser considerados. O uso de uma topologia como árvore e todos os nós conectados, muito comum em *datacenter* depende de diversos fatores e a visibilidade do experimentador pode não ser o suficiente para a correta interpretação dos dados.

Neste aspecto, emuladores não necessitam de grandes mudanças em questão de código ou configuração para um usuário, já que o ambiente emulado pode utilizar de recursos reais sem demais alterações, além de solucionar vários problemas de facilidade ao usuário (*frontend*, ambiente interativo e etc), a rapidez em criar um cenário emulado, com as características específicas para o experimentador, o tornam uma escolha tentadora, por ser flexível, com precisão e sem a necessidade de nenhum operador externo para que os experimentos aconteçam. Porém, da mesma forma que a *testbed*, para grandes cenários existem limitações da máquina

e dos emuladores, tornando a escalabilidade um problema, ainda não é possível emular links maiores do que já são apresentados, por serem recursos da máquina que o emulador utiliza para disponibilizar o ambiente, links de 10Gbps não são possíveis. Já através de simulação, por usar apenas algoritmos matemáticos para esse fim, limitações de máquina não são um problema. O cenário depende apenas dos objetos já modelados para o simulador e o nível de detalhes do experimentador ao criá-lo. A maior desvantagem dos simuladores é: caso o objeto ainda não esteja devidamente modelado, a complexidade em adicioná-lo pode aumentar o tempo necessário para uma tarefa que se tornaria mais simples em outros métodos, como emulador ou *testbed*, tornando-o inviável para alguns casos.

A simulação apesar de parecer automaticamente a com melhores resultados, necessita de grande entendimento sobre a plataforma a ser utilizada, além da compreensão direta sobre o objeto que pretende ser simulado. Essa curva criada entre a vantagem clara e a necessidade de conhecimento específico em programação pode dificultar a execução de testes básicos, o que diminui a atratividade do uso de simulações. Também não é possível ainda verificar, em uma máquina simulada, outros pontos como consumo de processamento e memória, o que descarta a tecnologia para o uso de testes com aplicativos mais simples, desenvolvidos justamente para avaliações de desempenho de máquina. Entende-se portanto que as restrições existem nas três tecnologias, cada qual com seus problemas e vantagens, apontando qual seria melhor para determinada execução de um teste.

2.3 Mininet

Dentre os diversos métodos disponíveis para pesquisa, a emulação como já dito é uma alternativa barata e que aproxima-se de resultados reais com grande facilidade de prototipação. Criado por professores de Stanford, o Mininet (LANTZ; HELLER; MCKEOWN, 2010) tem como principal objetivo facilitar o uso e o entendimento de tecnologias de rede de forma prática. A API de fácil utilização é um dínamo no uso da tecnologia, de fácil instalação e uso com CLI própria, a ferramenta tem fortes pontos positivos para uso básico e intermediário em topologias e pesquisa. A estrutura segue a ideia de utilizar a plataforma de containers do Linux, para emular ambientes isolados entre si e interconectados via OVS, tudo preparado de forma automática pela ferramenta ou totalmente específica pela API. De ambas as formas é possível interagir em tempo real, se desejado, de forma que o experimentador possa personalizar e atuar diretamente no ambiente (KULDEEP K. SHARMA, 2014).

2.3.1 Estrutura

Para melhor determinar o funcionamento é imprescindível lembrar que o Mininet utiliza a fórmula de *Container Based Emulation* e sua associação direta com *containers* em Linux. A figura a seguir demonstra em conjunto o conceito de *Containers* e Mininet:

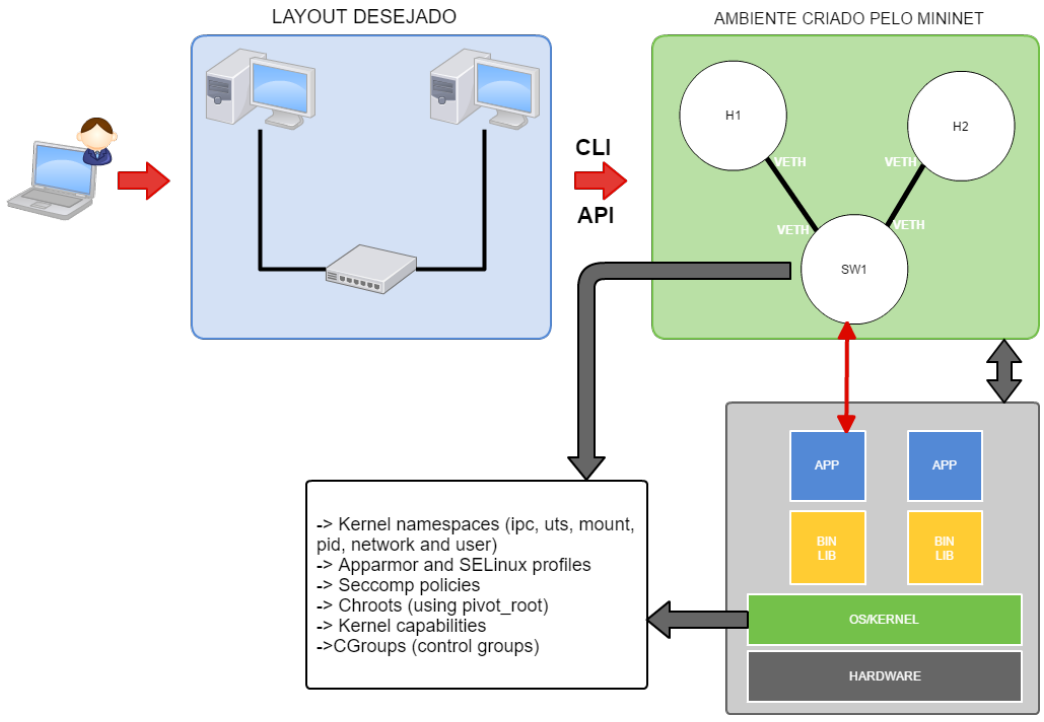

Figura 2.5: Emulação de ambiente via Mininet (LANTZ; HELLER; MCKEOWN, 2010) (KULDEEP K. SHARMA, 2014)

A figura representa como o Mininet utiliza de suas ferramentas para entregar o resultado final. Em um primeiro momento o experimentador escolhe, idealiza e prototipa layout desejado para sua pesquisa e, em seguida, escreve todo detalhamento, que será melhor abordado a seguir. Os *layouts* podem ser totalmente personalizados ou uma das estruturas que a ferramenta entrega. Cada nó idealizado é um *container* com todo isolamento provido pelo Linux e os APPs providos pelo Mininet, assim sendo, a camada de APP pode utilizar não só o que já é entregue, mas aplicações desenvolvidas para um fim específico, sendo um dos melhores exemplos, o uso do OVS como switch ou a troca para uma aplicação específica, apresentado em (ROTTENBERG,) com uma nova proposta de switch OpenFlow. As ligações entre *containers* são estabelecidas através de interfaces virtuais (VETH), sendo host ou switch. Normalmente existe um nó controlador para as funções OpenFlow disponíveis que, após o link VETH, pode ser feito em conexão segura como requerido por alguns controladores. Este pode estar contido em um nó ou no próprio ambiente do usuário, visto que é característica clara da emulação a possibilidade do uso de recursos e interligação com o computador servidor.

Essa estrutura de facilidades na qual toda necessidade de configuração é automatizada, entrega um ambiente de fácil utilização, visto que toda configuração de endereços, hosts, switches e links é feita de maneira transparente ao usuário. Os pontos.

2.3.2 Plataforma

A plataforma, apenas disponível em ambiente Linux, utiliza de recursos disponíveis do sistema operacional para poder construir o layout desejado. Para tal, o Mininet utiliza do recurso de containers presente no sistema operacional (JOY, 2015) para emular ambientes isolados, que representam separadamente cada um dos nós e switches, interconectados normalmente por interfaces virtuais.

Pós instalação do Mininet, feita de forma simples por hoje estar integrada aos principais repositórios, basta o experimentador idealizar o layout e utilizar das estruturas prontas via CLI ou personalizar via API.

Cada nó idealizado pelo experimentador será um container, com algumas das funções simples preparadas pela própria ferramenta. De forma inicial e sem necessidade nenhuma modificação ficam disponíveis para o uso diretamente pela CLI:

- help: apresenta todos os comandos disponíveis;
- nodes: exibe os nós emulados, tanto switches quanto hosts ou controladores;

- dump: lista as informações de todos os nós de forma detalhada;
- link: retorna quais hosts e switches estão interconectados;
- net: exibe todas as informações de rede do ambiente simulado, incluindo o mesmo resultado de link;
- ping: demonstra a conectividade entre dois hosts;
- pingall: testa a conectividade entre todos os hosts no ambiente simulado;
- pingallfull: mesma função do pingall, porém com mais detalhes ao final da execução;
- iperf: teste de banda entre hosts utilizando TCP;
- iperfudp: teste de banda entre hosts utilizando UDP;
- ifconfig: usado para checar o endereço IP de um host;
- xterm: abre uma janela separada conectada diretamente no nó;
- dpctl: mostra todos os fluxos OpenFlow instalados nos switches;
- exit: finaliza o ambiente de forma correta;

O uso pode se iniciar de forma simples em uma topologia mínima ou da personalização via API, ainda, como já citado, o uso comum do OVS com OpenFlow pode ser substituído por uma aplicação switch própria, como parte da estrutura de APP e *containers*. As topologias que seguem podem ser geradas automaticamente no Mininet a partir de linha de comando:

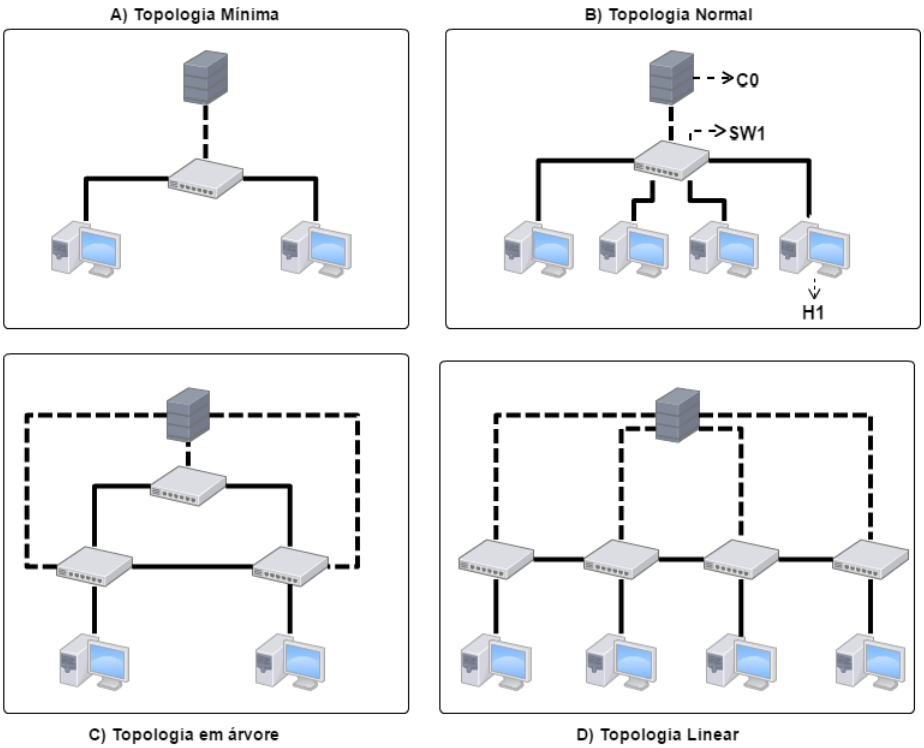

Figura 2.6: Topologias disponíveis (KULDEEP K. SHARMA, 2014).

As topologias podem ser utilizadas diretamente ao começar o Mininet:

- Topologia Mínima: Caso o usuário não especificar qual das topologias gostaria de utilizar, esse formato é o que a plataforma irá criar automaticamente;
- Topologia Normal: Composta por quatro hosts interligados em um único switch e controlador;
- Topologia Linear: Quatro switches interligados de forma linear com o mesmo controlador e um host por switch;
- Topologia em Árvore: Três switches interconectados entre si (*full*) com dois hosts em pontas;

As topologias podem ser utilizadas da forma a seguir, visto que existem características que podem ser customizadas diretamente via CLI:

```
sudo mn --topo=linear,\textbf{4} --links  tc,bw=10,delay=10ms
```

Desse modo as topologias customizadas diretamente pela CLI, apresentam características como, a quantidade de switches em uma topologia linear demonstrada no item D da figura 2.6.

A característica dos links pode ser definida como demonstrado, modificando a quantidade de banda para os links e o delay.

O uso de controladores pode ser feito de forma automática, na qual o OVS ou switch APP conectam em um controlador padrão com a função switch apenas ou um controlador externo, rodando as próprias aplicações:

```
sudo mn --controller=remote,ip=[controller IP],port=[controller listening port]
```

Todas as características acima definem diretamente a topologia que pode ser gerada pelo Mininet, mas existem formas de personalizar link a link, host a host e ligações específicas ou mesmo automatizar a execução de testes. A API do Mininet é rica em documentação e permite um controle refinado dentro dos parâmetros que a ferramenta entrega. Alguns itens que não são possíveis pela CLI podem ser alcançados diretamente pela API, como o controle de uso de processamento por cada host/nó, de forma que exista um controle de consumo entre os hosts. Lembrando que um ponto do uso CBE é o compartilhamento de recursos entre todos os nós emulados, algo que, dependendo da quantidade, pode influenciar no resultado final. Alguns dos pontos que podem ser customizados são (MININET..., b):

- Links: possível especificar a conexão entre os nós, bem como a banda, delay, jitter, perda, fila, latência;
- Node: possui métodos para controlar o consumo de processamento de cada nó, especificar um tipo de controlador como Ryu, Pox caso rode em um dos *containers* ou no ambiente do usuário (*RemoteController*), especificação do tipo de switch (OVS, Indigo ou do *userspace* do usuário);
- Net: todas as características de rede da emulação no namespace. Este objeto que concentra quase toda a emulação. *Pool* de endereços de rede IP, MAC e outras características podem ser setadas diretamente através da classe net;
- Log: métodos para acompanhar o log dos eventos, dados, links e qualquer item que execute na emulação;
- Topo: classe utilitária, que pode auxiliar na construção de topologias baseadas nas apresentadas na figura 2.6;
- Clean: finaliza todas as conexões e certifica-se de terminar todos os processos e *containers*;

O uso da API parte de um arquivo Python, desenvolvido baseado nas classes anteriormente citadas de acordo com a documentação (MININET..., b). O exemplo que segue cria uma estrutura simples, utilizando de alguns dos conceitos abordados:

```
from mininet.net import Mininet
from mininet.node import CPULimitedHost
from mininet.topolib import TreeTopo
from mininet.util import custom, waitListening, pmonitor
from mininet.log import setLogLevel, info

def bwtest( cpuLimits, period_us=100000, seconds=10 ):

topo = TreeTopo( depth=1, fanout=2 )

results = {}

for cpu in cpuLimits:
host = custom( CPULimitedHost, sched=sched, period_us=period_us, cpu=.5*cpu )
net = Mininet( topo=topo, host=host )
net.start()
net.pingAll()
hosts = [ net.getNodeByName( h ) for h in topo.hosts() ]
client, server = hosts[ 0 ], hosts[ -1 ]
info( '*** Starting iperf with %d%% of CPU allocated to hosts\n' % ( 100.0 * cpu )

net.stop()
updated = results.get( sched, [] )
updated += [ ( cpu, bps ) ]
results[ sched ] = updated

return results

def dump( results ):
"Dump results"

fmt = '%s\t%s\t%s\n'
```

```
info( '\n' )
info( fmt % ( 'sched', 'cpu', 'received bits/sec' ) )

for sched in sorted( results.keys() ):
entries = results[ sched ]
for cpu, bps in entries:
pct = '%d%%' % ( cpu * 100 )
mbps = '%.2e' % bps
info( fmt % ( sched, pct, mbps ) )

if __name__ == '__main__':
setLogLevel( 'info' )
out = bwtest( limits )
dump( out )
```

Observando o código, é possível visualizar que cada host tem sua limitação de CPU a partir da quantidade de hosts, os links tem sua limitação e a topologia executa e retorna automaticamente os resultados.

Essa seção busca apenas cobrir de forma genérica as muitas formas e qualidades que podem ser observadas no uso do Mininet, sem se aprofundar demais no desenvolvimento. Durante o capítulo de Arquitetura e Implementação os conceitos serão abordados com mais precisão de detalhes conforme o contexto.

2.4 Network Simulator - NS-3

O NS-3 é descrito como um simulador de eventos discretos, suportando pesquisas em redes IP e não IP, com código aberto, e de fácil depuração, que permite o desenvolvimento de simulações próximas do ambiente real. Quando começou a ser desenvolvido, alguns pontos foram considerados para manter a eficiência, independente da CPU e memória do computador que fosse executar a simulação:

- Suporte à fragmentação e remontagem: Tanto o protocolo IP quanto o MAC oferecem suporte à fragmentação dos dados a serem transmitidos (pacotes e quadros, respectivamente), que tem impacto no desempenho e comportamento final de uma aplicação. Com esse suporte, simulações envolvendo protocolos como *Multi-Path* TCP são possíveis.

- Dados reais de forma transparente: O simulador é capaz de emular e gerar pacotes fora de seu ambiente. Esse processo de conversão de pacotes simulados para reais é feita de forma transparente, sem a necessidade do usuário ou pesquisador ter que programar essa conversão.

- Dados exibidos de forma nítida: Para auxiliar o usuário ou pesquisador a melhor entender os resultados obtidos eles devem ser de fácil compreensão. Dessa forma, compreender os resultados ou entender o caminho de uma informação, se tornam processos mais rápidos e eficientes.

- Uso eficiente de memória: Alguns protocolos podem saturar a memória do computador que está executando a simulação, portanto o controle de uso da memória é importante. Dessa forma, simulações amplas ou com protocolos que utilizam muita memória não influenciam no resultado final.

- Uso eficiente de CPU: É imprescindível que o gerenciamento de pacotes não se torne um gargalo na CPU, prejudicando a simulação. Mecanismos de otimização do uso da CPU evitam esse tipo de problema em uma simulação.

- Suporte à extensão: É possível adicionar novos tipos de dados, cabeçalhos e protocolos sem ter que modificar o simulador por inteiro, possibilitando trabalhar com simulações que fogem de redes comuns. Um exemplo é a simulação de redes NDN (*Named Data Network*), já implementado (AFANASYEV, 2012).

2.4.1 Tratamento dos Dados

Uma questão importante para o simulador é como os dados devem ser tratados durante a simulação, uma das formas de tratamento, ”Simulação orientada a evento em tempo discreto”(CHONG, 1994), que tem como seu maior objetivo calcular o tempo total utilizado em cada *slot* de tempo. Nessa forma de simulação existem ainda outras duas técnicas, das quais o NS-3 utiliza apenas uma, tempo de execução orientado a evento. Essa técnica calcula apenas os valores de quando algo significativo acontece no sistema modelado, isso quer dizer que ao invés de calcular todos os componentes da simulação o tempo todo, se preocupa apenas naqueles que tem alguma interação. Para isso algumas ações são essenciais no funcionamento do simulador:

- Execução: Cada evento é executado em ordem crescente de tempo até que não haja mais nenhum ou que o usuário peça a interrupção do processamento, através do comando de Parada.

- Parada: Seta a variável global de parada para ”verdadeiro”. O processo de execução para e retorna os valores na oportunidade mais próxima do ciclo de execução.
- *Now*: Retorna o tempo de simulação naquele exato momento.
- Escalonador: Cria um novo evento, o disponibiliza na lista de eventos globais e retorna a referência para o novo evento criado.
- Cancelamento: Aciona uma *flag* de cancelamento no evento enquanto está na lista global. Essa *flag* é verificada pelo processo de Execução antes de executar o evento.
- Remover: Remove o evento da lista global, certificando de que nunca será executado. Essa operação sempre deve ser evitada por sua complexidade O(n), já que a lista deve ser percorrida por total a procura do evento a ser removido.
- *Status*: Retorna se um evento expirou ou se ele ainda está em processo de execução.

Essas ações completam todas as necessidades que o simulador tem durante a execução do modelo, no qual cada uma das ações permite um gerenciamento refinado sobre os componentes e sua execução, permitindo um alto controle, inclusive, durante o tempo de execução da simulação. As ações auxiliam também, de forma significativa, na coleta de informações pertinentes ao pesquisador, que pode controlar o evento com grande riqueza de detalhes.

2.4.2 Tempo de Simulação

O tempo pode ser calculado de diversas formas, um agente dependente do cálculo de tempo é o tipo de dado que será utilizado. A simulação de tempo no NS-3, utiliza uma integral de 64-bit que pode permitir uma precisão na casa de nanosegundos durante a execução da simulação. Todo tratamento de tempo no simulador é concentrado em uma classe chamada *Time*, que permite de forma transparente a conversão de segundos em nanosegundos ou microssegundos (dependendo da necessidade), além de prover a conversão implícita para a integral: int64x64t. Para isso, os operadores aritméticos (+,-,/,*) foram sobrecarregados de modo que possam trabalhar apenas com dados do tipo *integer*, impossibilitando os erros de métodos que trabalham com o tempo de execução. O tempo está fortemente associado a cada evento simulado, o que gera automaticamente todo código necessário através de modelos. O escalonador coleta diferentes tipos de variáveis, sendo a primeira, o atraso até que o evento expire, o segundo, um ponteiro ao método que deve ser chamado quando o evento expirar e o terceiro, o objeto em que o evento foi chamado.

2.4.3 Modelagem e Modularidade

O NS-3 deve ser capaz de otimizar os objetos de simulação ao ponto de aproximar do ambiente real, para isso alguns fatores devem ser considerados: Modularidade, é importante que o NS-3 seja modular mas não em todas suas camadas, isso pode prover um nível de abstração não desejado para a comunidade, em questão de nível de código e funcionamento do *core* NS-3; Configuração de parâmetros, para que permita o usuário ou pesquisador controlar uma gama de parâmetros e valores de forma fácil, como por exemplo, poder especificar o atraso de um *link* ou interromper a comunicação de algum objeto da simulação depois de X tempo percorrido; Coleta de dados, permite uma interpretação melhor do comportamento do ambiente com o que deve ser simulado, fica claro que de nada adianta permitir a simulação de um ambiente enorme e escalável se não for possível coletar dados suficientes.

A modularidade do ambiente NS-3 funciona com a combinação de diferentes objetos para a simulação como demonstra a figura 2.7:

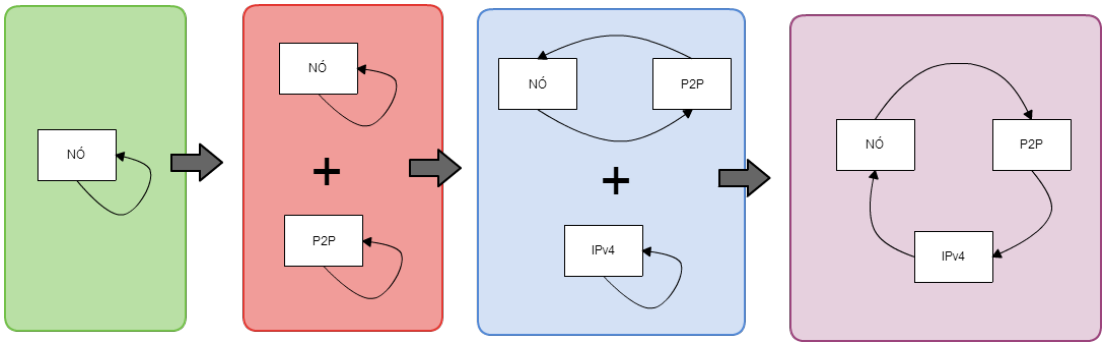

Figura 2.7: Modularização

Neste exemplo, o objeto nó, é combinado com o modelo de comunicação *P2P*, mas sabemos que protocolo IPv4 é necessário, com o qual é combinado logo em seguida e assim sucessivamente com todos os módulos necessários para o funcionamento de um objeto. O mesmo vale para todos os tipos de simulações que, inclusive, podem utilizar implementações próprias de outros protocolos, modelos de ambiente e funcionamento de nós específicos. Sendo assim, a forma modular de combinação de objetos, permite que muitos testes a nível de usuário possam ser realizados com implementações próprias, sem interferir ou modificar a estrutura de funcionamento e desenvolvimento do NS-3.

Os parâmetros apontados anteriormente são em sua maioria providos pelo próprio simulador, permitindo um controle abrangente sobre os objetos envolvidos na simulação. A classe Objeto, que forma cada um dos componentes do modelo de simulação, é responsável por todos os parâmetros que podem ser modificados em tempo de execução, centralizando todos os

dados de cada possível subclasse do objeto. Esses parâmetros podem modificar um atributo diretamente em um objeto qualquer (desde que ele possua suporte para isso, como dito anteriormente), isso feito através de duas formas mais específicas:

- Linha de Comando: Uma classe que reconhece toda lista de comandos possíveis, em um objeto que possa alterar algum atributo. Um exemplo é modificar o atraso de um *link*, a verificação de *checksum* nos pacotes, dentre vários outros.
- Repositório de Configurações: Busca dentro do objeto todas as modificações possíveis e as concentra em um arquivo XML ou *raw*. Esse mesmo tipo de arquivo pode ser utilizado para que, ao invés de demonstrar todas as ações possíveis em um objeto, rode novamente a simulação com tipos e dados determinados para cada atributo.

Um recurso presente em todo código produzido do NS-3 é o uso da ferramenta *Doxygen* (DOXYGEN,) para gerar a documentação, ao produzir código para o NS-3 existe uma forte recomendação, no caso de ser submetido ao projeto principal, em adequar o código a essa ferramenta.

2.4.4 Rastreamento dos Dados

Além de gerar as simulações, é importante que exista um bom sistema de *tracing* das informações relacionadas aos eventos em execução. O NS-3 possui uma gama de recursos de rastreamento que possibilitam, o usuário ou pesquisador, gerar dados o suficiente para compreender as ações e seus resultados durante a execução da simulação. Existem alguns pontos básicos do sistema de rastreamento presente no NS-3 que são:

- Os usuários podem estender o sistema de rastreamento para modificar os formatos da saída gerada ou inserir novas fontes fontes de rastreamento sem modificar o núcleo do simulador, permitindo a saída nítida das informações geradas ajustadas à necessidade do usuário ou pesquisador. Isso habilita o NS-3 a exibir o conteúdo final de forma amigável e apenas com dados que sejam pertinentes, eliminando o excesso de informações que não apetecem à pesquisa.
- Um sistema padrão, que fornece o rastreamento através de partes conhecidas e a customização dos objetos que o geram.
- Usuários podem modificar o núcleo do simulador para adicionar novas origens e destinos do rastreamento. Isso permite que em casos de usos não comum, como o já citado NDN, possam ser criados sistemas que se adequem melhor às necessidades não padrões.

- É possível adicionar *TAGS* específicas em cada pacote, como o tempo de simulação auxiliando a inserção de informações pertinentes em cada pacote simulado.

- Possui uma lista auxiliar separada de cabeçalhos e rodapés para verificação do usuário diretamente na memória sem interromper ou interferir em algum ponto da simulação.

Para este sistema de rastreamento existem níveis de interação com a coleta de dados pertinentes a cada evento. Podem ir de mensagens de depuração até o registro total de todas as informações que correm no simulador. Esses níveis podem ser definidos no modelo da simulação em âmbito global (para todos os eventos) ou em eventos e nós específicos. Esses níveis tem características pessoais de registro e podem ser definidos dinamicamente, utilizando o recurso de linha de comando já apresentado. Os níveis são divididos em: *WARN*, registra apenas mensagens de alerta; *DEBUG*, mensagens mais raras; *INFO*, registra mensagens informativas sobre o progresso da simulação ou evento em particular; *FUNCTION*, registra as mensagens descrevendo cada função de chamada; *LOGIC*, coleta apenas mensagens que possam descrever o fluxo lógico dentro de uma função; *ALL*, registra todas as mensagens e por fim; *UNCOND*, que diferente do tipo *ALL*, registra todas as mensagens que ocorrem incondicionalmente, o que pode ser útil para procurar por erros no modelo ou comunicação em algum ponto. O sistema funciona de forma bem simples para que possa ter toda essa robustez, cada origem de rastreamento é uma entidade que pode assinalar eventos que ocorrem em uma simulação e fornecem acesso a dados de baixo nível e do outro lado existem entidades que consomem essas informações. Simplificando, origens de rastreamento (*trace source*) são geradores de informações enquanto o destino do rastreamento são consumidores de informação (*trace sinks*), dessa forma podem existir diversos pontos de origem com um único ponto de destino.

Para que os pontos de rastreamento sejam bem definidos é necessário modelar a simulação de forma correta. Alguns modelos já estão implementados no NS-3, cobrindo grande parte das camadas de rede necessárias: pilhas IPv4, ARP, UDP e TCP. Essas pilhas podem integrar as implementações reais em Linux e BSD utilizando o *Network Simulatuion Craddle* (TAZAKI; URBANI; TURLETTI, 2013), aumentando o nível de aproximação a um ambiente real nos resultados. As camadas devem ser adicionadas manualmente, o que aumentaria a complexidade de desenvolvimento mas, para este fim os desenvolvedores do NS-3 disponibilizaram uma forma bem mais simples de adicionar as pilhas já implementadas nos modelos de simulação chamados de *helper*. A figura 2.8 a seguir demonstra a organização do *software*:

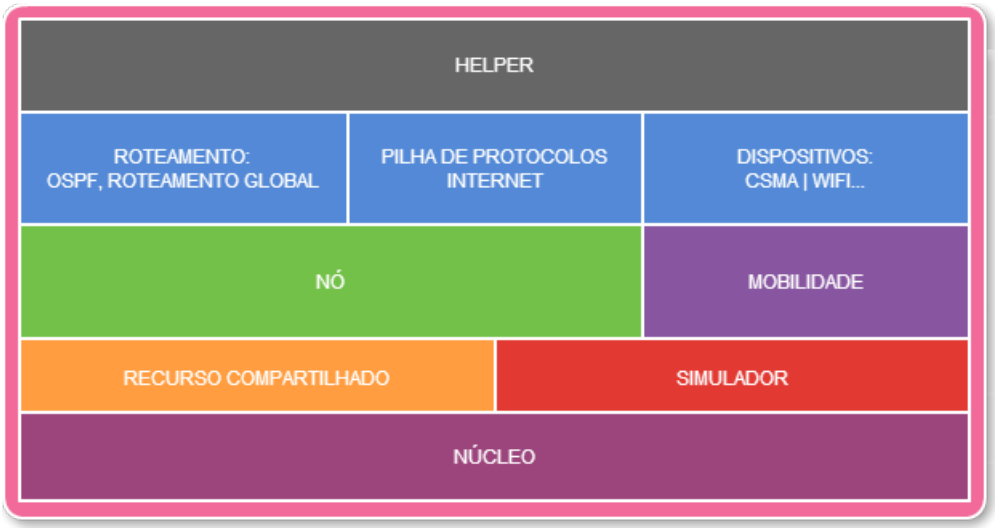

Figura 2.8: Divisão de módulos do NS-3

2.4.5 Tratamento dos dados

Um Framework que utiliza estruturas em memória para armazenar os resultados, sendo possível salvar em disco em arquivos XML. O FlowMonitor(CARNEIRO; FORTUNA; RICARDO, 2009) é escrito em Python e C++, unindo os dois através do GIL (Global Interpreter Lock), o qual converte os parâmetros em C++ para Python e converte os valores de retorno de Python para C++. Como Python é uma linguagem de script um tanto lenta, uma classe C++ faz o tracing e devolve os resultados para o código Python.

Sua estrutura é composta pelas seguintes classes:

- FlowMonitor: coordena os probes da rede;
- FlowProbe: escuta os eventos em um ponto específico da rede simulada, reporta eventos para o FlowMonitor e colhe as estatísticas de fluxo referentes ao seu probe;
- FlowClassifier: classifica o fluxo através de um número de 32 bits, que identifica um pacote dentro de determinado fluxo;

Os principais resultados são armazenados em estruturas na memória, acessíveis por métodos getter. Para estatísticas, podemos chamar FlowMonitor::FlowStats e FlowProbe::FlowStats, sendo o último, um pequeno conjunto das estatísticas referentes somente ao seu probe. Ao chamar FlowMonitor::FlowStats, temos os seguintes atributos disponíveis:

- timeFirstTxPacket: horário de inicio de transmissão do fluxo;
- timeLastTxPacket: horário de término de transmissão do fluxo;

- timeFirstRxPacket: horário de recebimento do primeiro pacote até um ponto final de um fluxo;
- timeFirstRxPacket: horário de recebimento do último pacote de um fluxo em um ponto final;
- delaySum: soma de todos os delays de um fluxo;
- jitterSum: soma de todo delay jitter de um fluxo;
- txBytes, txPackets: número total de, respectivamente, bytes e pacotes transmitidos por um fluxo;
- rxBytes, rxPackets: número total de, respectivamente, bytes e pacotes recebidos por um fluxo;
- lostPackets: total de pacotes perdidos em um fluxo;
- delayHistogram, jitterHistogram, PacketHistogram: histograma dessas informações;

Como dados de jitter, delay etc são muito sensíveis para apenas fazer uma soma dos mesmos, o framework utiliza histogramas, em vez de armazenar os resultados em memória (o que seria, em termos computacionais, atualmente impossível, por conta da quantidade de informações.

2.4.6 Fluxo de Desenvolvimento

Os *helpers* auxiliam o usuário a adicionar essas pilhas e objetos em seu modelo de simulação com *scripts* transparentes ou seja, não é necessário implementar a pilha do protocolo *Ethernet* por exemplo, o *helper* adiciona todas as necessidades para o uso automaticamente. Essa funcionalidade foi implementada também para endereçamento IPv4, roteamento L3, configuração de dispositivos e nós, além da facilidade na utilização das pilhas reais do Linux. Ainda na imagem 2.8 os nós devem conter todas as pilhas necessárias para seu funcionamento, como um dispositivo real, por isso, parte da configuração de um nó pode ser facilitada pelo *InternetStack*, que já adiciona boa parte das funcionalidades necessárias. A mobilidade adiciona recursos de testes para tecnologias sem fio (*WiFi*, *Wi MAX*, redes de sensores), a partir das quais é possível simular um dispositivo em movimento para simulação. O simulador em si é responsável por toda parte de agendamento, tempo e administração dos eventos, enquanto o compartilhado trata todos os aspectos de pacotes, inclusive o gerador de arquivos *pcap* pós execução. O núcleo, por fim, é responsável por todo *callback*, *tracing*, ponteiro, dentre outras necessidades de sistema geradas pelo NS-3.

Nesse contexto, criar os modelos segue o fluxo demonstrado pela figura 2.9:

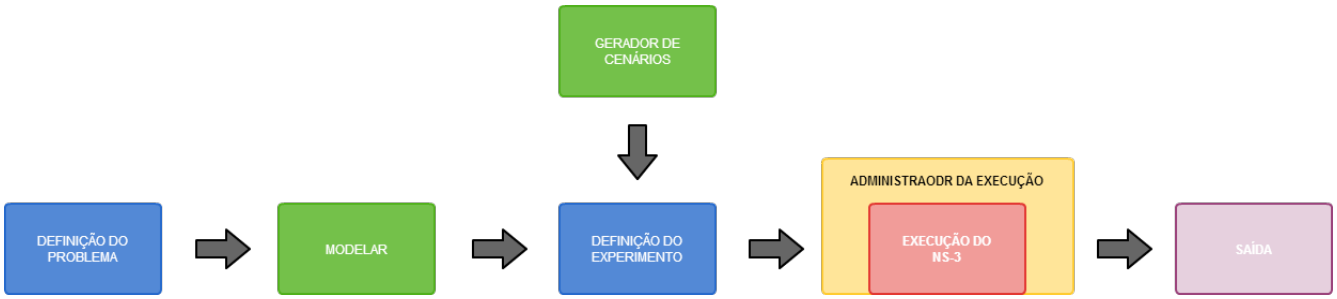

Figura 2.9: Fluxo de Desenvolvimento: Da abstração à coleta de resultados.

Fica claro que entender bem o problema, é essencial para uma boa modelagem da simulação. O cenário é gerado a partir da modelagem inicial, junto com as definições de experimentação, para que então a simulação seja executada e ao término dessa execução, temos as saídas com os rastreamentos que foram pedidos, bem como arquivo *pcap* com os dados de pacotes que trafegaram. Para que este *pcap* seja gerado, o NS-3 possui um tratamento para que os pacotes possam se aproximar do tratamento de pacotes reais, aproximando a fidelidade do simulador a um sistema não simulado. Alguns aspectos são importantes no tratamento de pacotes dentro da simulação:

- Recepção de pacote: Quando chega ao *buffer* nunca será escrito diretamente. Cada camada apenas incrementa o que é necessário e repassa para a próxima camada o mais rápido possível sem se importar com conteúdo do pacote.
- Repasse de pacote: Sempre começa com a recepção do pacote, mesmo quando existe a necessidade de troca de pacotes entre nós na rede. Uma implementação mais robusta faz uma cópia do pacote antes de enviar para as camadas mais abaixo, evitando qualquer problema com a decisão do repasse.
- Retransmissão de pacote: Quando um pacote simplesmente passa pela pilha, o cabeçalho é adicionado ou retirado. Quando uma camada precisa de confirmação do recebido (no caso de entrega do TCP por exemplo), é necessário que faça uma cópia do pacote antes de ser enviado nas camadas abaixo, como é citado no *buffer* do repasse de pacotes.

A implementação desse tratamento de pacotes envolve, inclusive, como eles serão gerados do ambiente simulado para o ambiente real de forma transparente ao usuário e vice versa. A implementação que torna possível essa realidade de conversão de pacotes se dá pelo *buffer*, que copia o pacote completamente e espera pelo modelo da simulação ler os cabeçalhos e rodapés

na ordem correta, sendo assim um cabeçalho não representa os campos iniciais de um pacote e sim o próprio pacote inicial de uma transmissão, além do rodapé final.

O tratamento de pacotes no NS-3 só é possível por algumas implementações, com base na falta de recursos disponibilizados por outros simuladores. O *payload* é representado pela *zero area*, um espaço virtual que não ocupa memória, sendo preenchido por zeros pela aplicação. Isso faz com que o custo de uso de memória de um pacote NS-3 seja reduzido, sendo um dos pontos de otimização do uso da memória e processamento citado anteriormente, já que quanto menor o pacote, menor o custo para copiar entre posições na memória. Parte dessa otimização do uso de memória e processamento acontece pela técnica de *COW* (*Copy-On-Right*), na qual dois objetos não relacionados entre si (a não ser pela comunicação), acessam uma área da memória compartilhada que possui a cópia do pacote completo com todas as informações de todas as camadas.

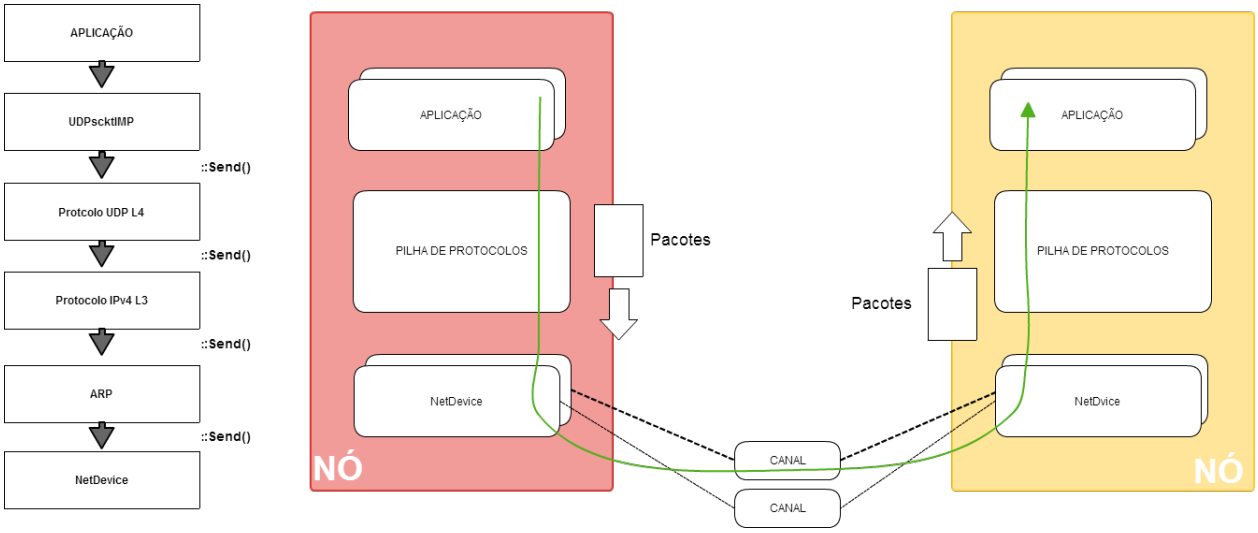

Figura 2.10: Caminho da aplicação

As figuras 2.11 e 2.12, representam respectivamente o pacote na memória compartilhada. É possível observar que, quando o cabeçalho é inserido, para que não haja um problema de referência entre os pacotes compartilhados, o pacote é marcado, separado e por fim não mais compartilhado na memória. Isso acontece para se evitar problemas recorrentes da leitura.

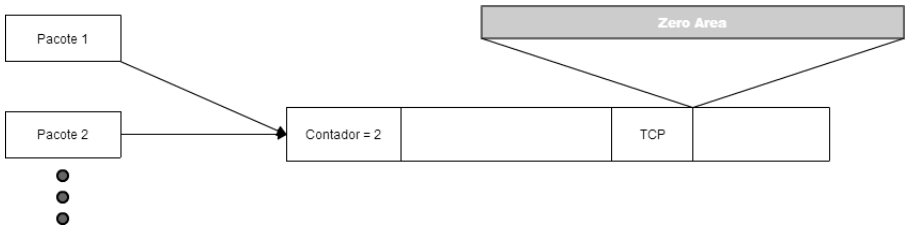

Figura 2.11: Memória compartilhada entre *hosts*.

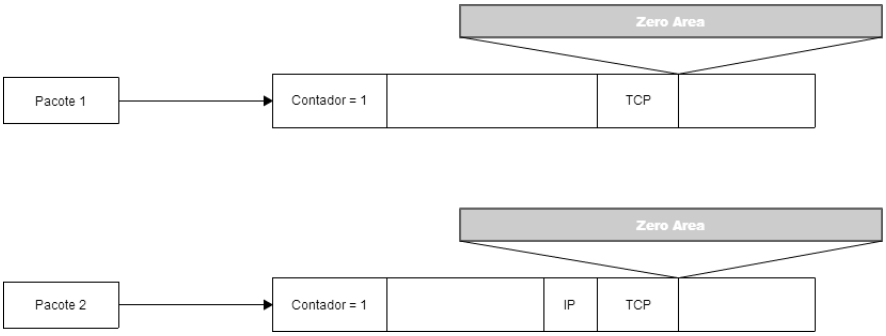

Figura 2.12: Memória deixando de ser compartilhada entre *hosts*.

Um exemplo da necessidade dessa separação, é um nó que tenta inserir um cabeçalho IP sobre uma cópia da memória que já possui um cabeçalho, causando portanto o não funcionamento da simulação. Existem mecanismos internos que mantém uma lista de modificações feitas no pacote que está na memória compartilhada. A figura 2.13 demonstra a comunicação entre dois pontos de forma completa durante a simulação, levando em conta todos os itens modelados do problema real e o fluxo de desenvolvimento da figura 2.9.

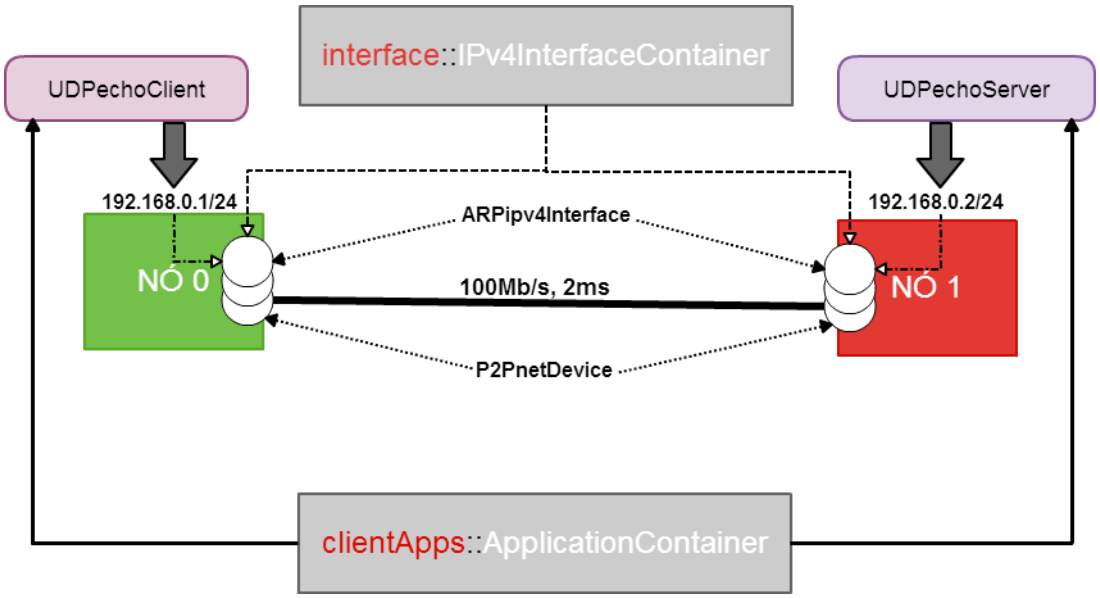

Figura 2.13: Abstração da comunicação usando contêineres.

Para que o modelo seja de fácil desenvolvimento existe o recurso de *conteiner*, que auxilia na administração de nós que pertencem a mesma classe, ou seja, podem existir contêineres de nós, *hosts* e aplicações. Isso facilita não só na administração e organização, mas também a configurar múltiplos objetos que utilizarão recursos similares. A figura 2.13 demonstra a forma de organização desses nós, no mesmo contêiner e a utilização de aplicações parecidas, também aglomeradas no mesmo contêiner, mas como pode ser visto, com funcionalidades diferentes.

O código a seguir representa, em modelo do NS-3, a mesma aplicação demonstrada na

figura 2.10. Os comentários pertinentes ao código demonstrando os passos de funcionalidades estão junto ao código:

```
17 #include "ns3/core-module.h"
18 #include "ns3/network-module.h"
19 #include "ns3/internet-module.h"
20 #include "ns3/point-to-point-module.h"
21 #include "ns3/applications-module.h"
22
23 using namespace ns3;
24
25 NS_LOG_COMPONENT_DEFINE ("Exemplo");
26
27 int
28 main (int argc, char *argv[])
29 {
30
34   NodeContainer nodes; //Cria o conteines de nós.
35   nodes.Create (2);    //Adiciona dois nós ao conteiner.
36
37   PointToPointHelper pointToPoint;  //Cria o Helper de conexão p2p
38   pointToPoint.SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
//Seta a velocidade de comunicação no canal
39   pointToPoint.SetChannelAttribute ("Delay", StringValue ("2ms")); //O delay
40
41   NetDeviceContainer devices; //Conteiner de dispositivos
42   devices = pointToPoint.Install (nodes); //Cria o link entre os dispositivos
43
44   InternetStackHelper stack; //Helper que contém todas as pilhas de protocolo
45   stack.Install (nodes); //Adiciona esse stack aos nós do conteiner de nós, t
46
47   Ipv4AddressHelper address; //Um Helper de endereçamento IPv4
48   address.SetBase ("10.1.1.0", "255.255.255.0"); //Base de endereçamento
49
50   Ipv4InterfaceContainer interfaces = address.Assign (devices); //Adiciona a
51
```

```
52   UdpEchoServerHelper echoServer (interfaces.GetAddress (1), 9); //Helper de
53
54   ApplicationContainer serverApps = echoServer.Install (nodes.Get (1)); //Cri
55   serverApps.Start (Seconds (1.0)); //Inicia a aplicação servidor
56   serverApps.Stop (Seconds (10.0)); //Para a aplicação servidor
57
58   UdpEchoClientHelper echoClient (interfaces.GetAddress (1), 9); Helper de Ec
59   echoClient.SetAttribute ("MaxPackets", UintegerValue (1)); //Atributo do cl
60   echoClient.SetAttribute ("Interval", TimeValue (Seconds (1.0))); //Atributo
61   echoClient.SetAttribute ("PacketSize", UintegerValue (1024)); //Atributo do
62
63   ApplicationContainer clientApps = echoClient.Install (nodes.Get (0)); //Cri
64   clientApps.Start (Seconds (2.0)); //Inicia a aplicação client
65   clientApps.Stop (Seconds (10.0)); //Para a aplicação client
66
67   Simulator::Run (); //Iinicia a simulação
68   Simulator::Destroy (); //Para a simução
69   return 0;
70 }
```

O resultado do modelo de simulação acima, apresentado com os níveis de rastreamento gera a seguinte saída final, levando em conta todos os tratamentos de saída:

```
Sent 1024 bytes to 10.1.2.4
Received 1024 bytes from 10.1.1.1
Received 1024 bytes from 10.1.2.4
```

2.4.7 NS-3 e OpenFlow

Atualmente existe uma solução OpenFlow para NS-3 adaptada totalmente para o simulador (NS-3..., b). Esse recurso ainda permanece na versão 0.8.9 (antes da recomendação de implementação da versão 1.0) do protocolo OpenFlow(CONSORTIUM et al., 2009), pela complexidade de portar qualquer novo recurso para o NS-3. Nesse modelo não é possível adicionar nenhum controlador externo, o que impossibilita o usuário testar diferentes aplicações desenvolvidas. Existem apenas duas alternativas que são: um controlador do tipo L2 *learning switch*

e um controlador que descarta todos os pacotes como regra padrão até outras sejam adicionadas manualmente, tornando complexo portar uma solução própria para o ambiente simulado. Dessa forma, seria necessário recriar o controlador a ser testado com todas as suas regras dentro do ambiente do NS-3, inviabilizando a facilidade de uso e as múltiplas opções de controladores para testes.

2.5 Trabalhos relacionados

Essa seção compreende um levantamento de estudos com simuladores e simulações que envolvem aspectos pertinentes à comparação e aprimoramento da proposta dessa tese. Todos os artigos a seguir, possuem informações que agregam valor à simulação ou utilizam o protocolo OpenFlow, demonstrando resultados e forma que foi implementado.

2.5.1 Direct Code Execution - DCE

Simuladores tem uma grande escalabilidade e seu nível de realismo e a complexidade de implementação de novos recursos é grande. Muitos recursos reais que são portados para simuladores não são continuados pelos desenvolvedores justamente pela complexidade em manter atualizado no simulador todas as alterações que acontecem. Para atingir o realismo necessário sem essa complexidade de implementação, o simulador deve ter a funcionalidade de permitir que códigos já implementados sejam parte da simulação, sem a necessidade de adequação de código para a arquitetura do simulador, usando de recursos da máquina do usuário sem quebrar as premissas de simulação. Desde que o tempo de simulação seja o mesmo para os recursos utilizados dentro e fora do simulador, o resultado com algo adaptado e outro não adaptado para o simulador é o mesmo.

Um recurso recentemente desenvolvido para o NS-3 tem exatamente essa funcionalidade: Inserir implementações do ambiente do usuário para o simulador aproximando do realismo e diminuindo a complexidade de execução. O DCE é um ambiente integrado ao NS-3, que permite utilizar de funcionalidades sem modificar nenhum aspecto em ponto de vista do usuário ou do simulador. Dessa forma, toda pilha de protocolos antes utilizada pelas implementações adaptadas para o simulador podem, nesse contexto, executar as pilhas do *kernel* diretamente. Isso adiciona escalabilidade e facilita manter as simulações atualizadas às modificações de atualização de um ambiente real, de outra forma seria necessário toda vez que um novo recurso ou atualização fosse feita no sistema, essa teria de ser estudada e portada para o simulador, o DCE quebra essa necessidade mantendo as premissas de simulação. Para que isso seja feito de forma que respeite

o ambiente simulado, alguns pontos precisam ser esclarecidos:

- Sem modificações no código fonte de ambos os lados, para que não haja dependência e mudanças regulares.
- É necessário que mesmo utilizando o ambiente fora do simulador, seja possível depurar a simulação como um todo.
- Que ainda exista eficiência de memória e processamento.
- Que seja possível incorporar no simulador as implementações do ambiente do *kernel* (como a pilha TCP/IP) e do ambiente de usuário (*Zebra daemon*).

O DCE utiliza de dois recursos para manter os aspectos acima citados: Virtualização do ambiente do Usuário (*user-space*) e virtualização do Kernel do Linux(*Kernel-space*).

2.5.2 High-fidelity switch models

Tanto o ambiente simulado quanto o emulado tentam ao máximo se aproximar de cenário real. Este artigo (HUANG; YOCUM; SNOEREN, 2013) adiciona um ponto muito pertinente ao pensamento: uma comparação de recursos e desempenho de *switches* OpenFlow reais e o OVS(OPEN...,) como NFV. O fato é que, enquanto os *switches* tem recurso limitado de processamento e memória, o OVS tem a sua disposição tudo que a máquina hospedeira pode oferecer, ou seja, enquanto o *switch* OpenFlow real tem sua tabela de fluxos alocada em uma memória limitada e seu processamento fixo, o OVS pode alcançar resultados muito superiores por não ter essa limitação, como pode ser observado na figura 2.14:

Switch	Firmware	CPU	Link	Flow table entries	Flow setup rate	Software table
HP ProCurve J9451A	K.15.06.5008	666 MHz	1 Gbps	1500	40 flows/sec	Yes
Fulcrum Monaco Reference	Open vSwitch 1.0.1	792 MHz	10 Gbps	511	42 flows/sec	No
Quanta LB4G	indigo-1.0-web-lb4g-rc3	825 MHz	1 Gbps	1912	38 flows/sec	Yes
Open vSwitch on Xeon X3210	version 1.7.0	2.13 GHz	1 Gbps	65,000	408 flows/sec	Only

Figura 2.14: Características de comparação entre *swtiches* reais e o NFV OVS

Com essa diferença de capacidade, resultados totalmente diferentes (sejam em simulações ou emulações) são apresentados. O artigo aqui discutido adiciona um *proxy* entre o controlador e o OVS para aumentar o *delay* de comunicação como mostra a figura 2.15:

Os resultados apresentados pela solução ajudaram o OVS a se aproximar de um *switch* real, aumentando a fidelidade de resultados e elegendo-o como um valioso recurso a ser utilizado,

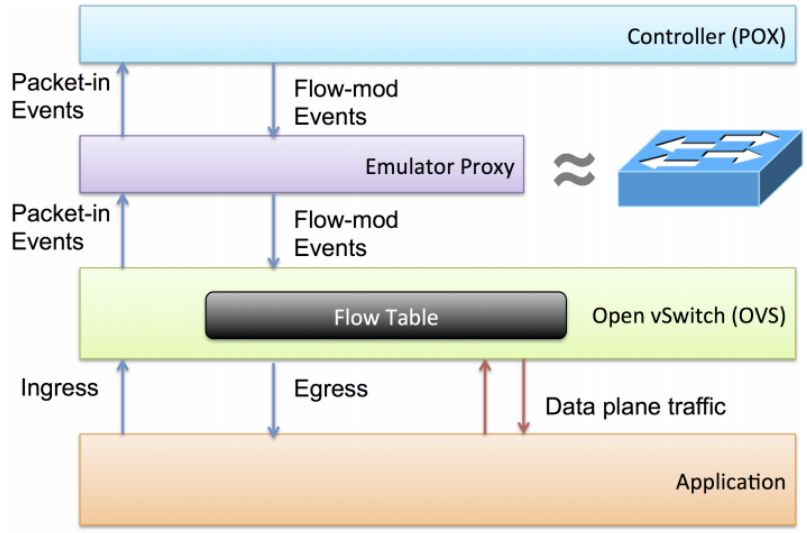

Figura 2.15: Solução Apresentada

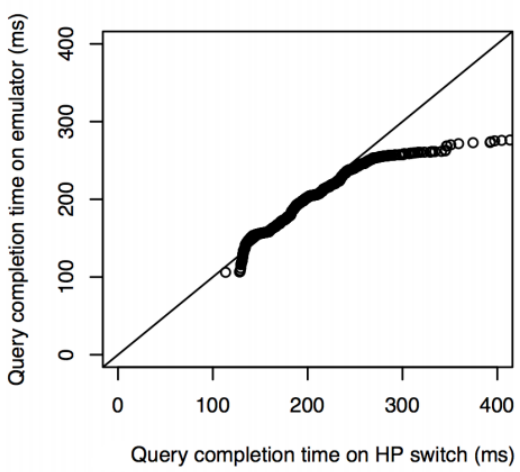

Figura 2.16: Comparação entre OVS e *switch* HP apresentado em 2.14

demonstrado na figura 2.14, sendo que a proximidade ao traço central é a igualdade entre os valores apresentados por ambos:

Mesmo com uma pequena variação, grande parte do resultado permanece próximo à linha central. Apesar dessas informações, o OVS ainda possui uma gama de recursos que não são interessantes ao experimento OpenFlow, além de ser bem mais complexo de alterar qualquer parte de seu código devido essas funcionalidades, dificultando a utilização em um ambiente simulado, tanto para modelagem quanto uso híbrido (Simuação/Emulação).

2.5.3 OpenFlow Controllers over EstiNet Network Simulator and Emulator: Functional Validation and Performance Evaluation

Este artigo compreende uma comparação entre o simulador com recurso de emulação, proprietário chamado EstiNet(TECHNOLOGIES,). Sua arquitetura permite que controladores externos sejam utilizados durante a simulação de redes SDN. A figura 2.17 resume todo funcionamento de uma simulação OpenFlow nesta plataforma:

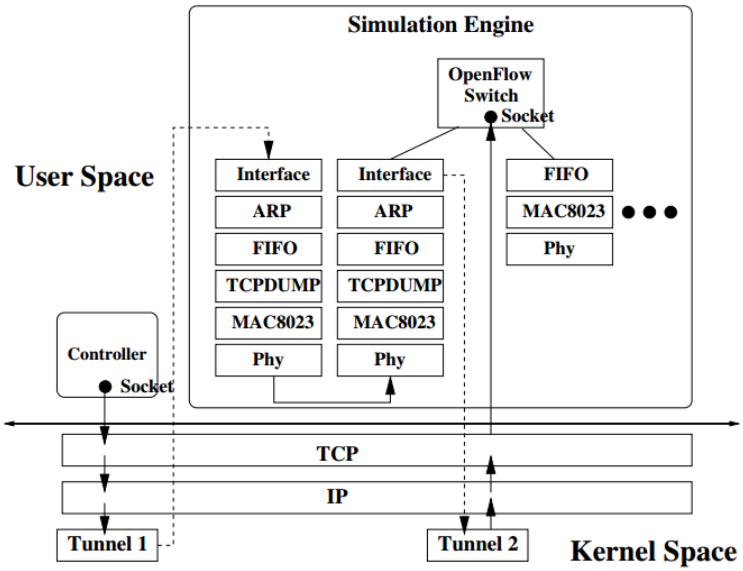

Figura 2.17: Arquitetura do Simulador EstiNet.

Seu funcionamento, não muito diferente do DCE(JANSEN; MCGREGOR, 2005), garante o uso do protocolo de forma escalável com controladores externos, sejam quais forem, indiferente de sua implementação. Este tipo de ferramenta oferece bem mais do que apenas recursos amplos, permite que o pesquisador usufrua de diferentes topologias para testes de sua solução. O artigo ainda apresenta a figura 2.18 de forma comparativa:

Fica claro que o NS-3 aparece como uma boa solução, porém defasada. Grande parte do objetivo desse trabalho pode ser visto aqui, na atualização do protocolo OpenFlow e inserção de recursos emulados OpenFlow de forma fácil para o usuário.

2.5.4 Fast, accurate simulation for SDN prototyping

Desenvolvido pela Universidade de Winsconsin baseado no simulador fs (SOMMERS, 2011), o fs-sdn (GUPTA; SOMMERS; BARFORD, 2013) adiciona o recurso de simulação OpenFlow com a premissa de fornecer grande escalabilidade sem afetar o realismo, balanceando ambos. Utiliza

	EstiNet	ns-3	Mininet
OpenFlow Specification	1.1.0/1.0.0	0.8.9	1.0.0
Simulation Mode	✓	✓	-
Emulation Mode	✓	-	✓
Compatible with Real-World Controllers	✓	-	✓
Result Repeatable	✓	✓	-
Scalability	**High** by Single Process	**High** by Single Process	**Middle** by Multiple Processes
Performance Result Correctness	✓	**No spanning tree protocol and no real-world controller**	**No performance fidelity**
GUI Support	✓ • Configuration • Observation	✓ • Observation Only • Using C++ for Configuration	✓ • Observation Only • Using python for Configuration

Figura 2.18: Comparação entre recursos.

de captura de fluxos da simulação em um grande nível de detalhe como sua principal característica e promete, ainda, uma grande facilidade no manuseio. Baseado em Python, também desenvolvida com o uso de DES (CHONG, 1994) para que o cálculo de seus resultados tenham a possibilidade de simular várias condições de tráfego, que podem ser configurados através de um JSON e não trata de simular cada pacote, mas usa de uma abstração mais complexa chamada de *flowlet*, que representa o volume de fluxo emitido em um determinado espaço de tempo, por exemplo: em 100ms de simulação, um *flowlet* pode conter um ou mais pacotes garantindo um desempenho superior ao NS-3 e a entrega de centenas de *switches* e *hosts* em seu ambiente simulado. Porém, mesmo com a premissa de uma grande escalabilidade, utilizada no artigo para recriar ambientes de *Data Centers*, é necessário que o controlador OpenFlow seja, obrigatoriamente POX(NOXREPO.ORG,), o que limita os recursos de desenvolvimento do pesquisador. Hoje com diversos controladores e sendo o mais famoso o Ryu (RYU,), mesmo com o suporte ao controlador externo à simulação, estar preso apenas a um tipo de controlador não o torna interessante para diversas pesquisas, ainda, algumas modificações são necessárias para que o controlador possa funcionar diretamente, de forma não transparente ao usuário anulando a premissa de fácil utilização.

2.5.5 An OpenFlow Extension for the OMNeT++ INET Framework

O simulador OMNeT++ (OMNET,) é famoso por suas características, mas sem suporte a OpenFlow. Esse artigo (KLEIN; JARSCHEL, 2013) demonstra a forma que foi incluído o suporte ao protocolo e alguns testes em cenários de grande escala. Para a utilização do OpenFlow foram inseridas: uma classe OFP Header, na qual cada subclasse representa um tipo de mensagem e comunicação OpenFlow, seja do controlador para o *switch* ou vice-versa; dois modelos, um representando o *switch* OpenFlow (2.19 B) e outro representando o controlador (2.19 A), mas ambos desenvolvidos dentro do simulador como segue:

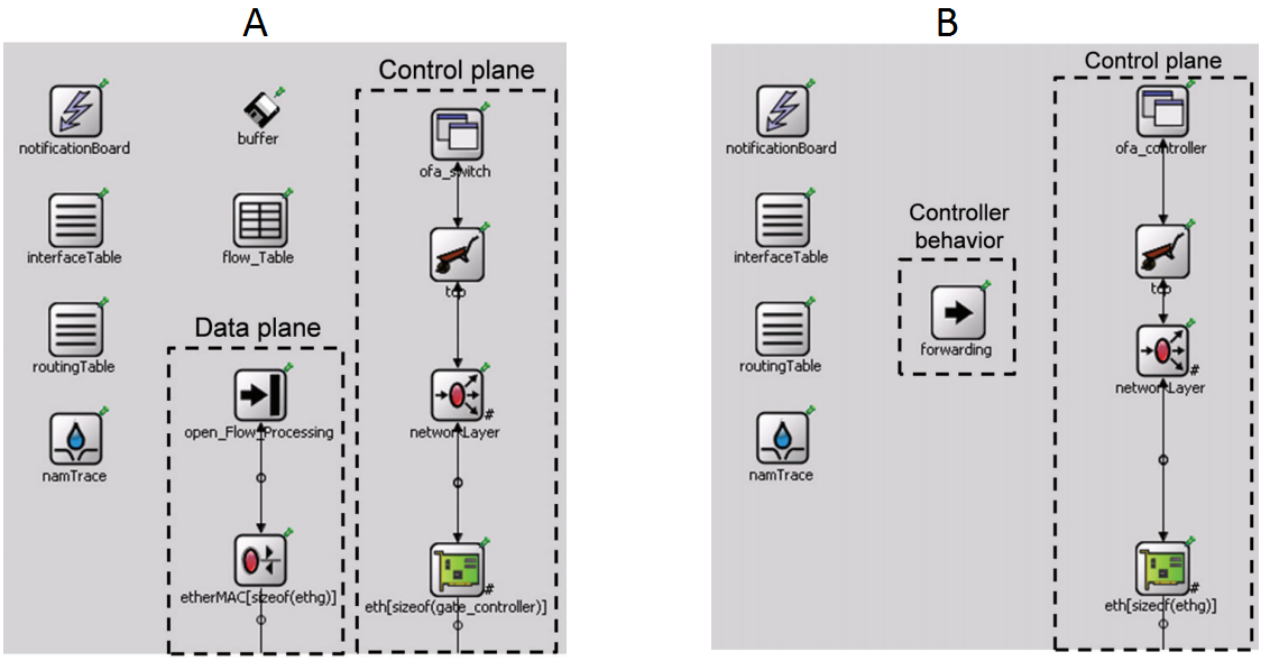

Figura 2.19: Modelo de recursos OpenFlow utilizados pelo OMNeT. Parte A representando o Controlador OpenFlow e parte B o *switch* OpenFlow

Essa forma de desenvolvimento apesar de permitir o controle da comunicação entre *switch* e o controlador, tratando parte do que o artigo propõe, não é possível moldar, por exemplo, o consumo de memória da *FlowTable* ou limitar o processamento. Ainda, essa forma de desenvolvimento fechada, obriga que as regras do controlador sejam desenvolvidas dentro do simulador, não permitindo uma comunicação externa, o que torna inviável para utilização de um desenvolvimento próprio. Apesar do simulador permitir e o artigo demonstrar a utilização com um ou demais controladores em larga escala, representando 34 *switches* com alguma fidelidade nos resultados, de nada vale esse recurso se o simulador obriga que toda lógica seja desenvolvida dentro da plataforma. Uma vantagem é a versão presente, OpenFlow 1.2, mas sem a possibilidade de um controlador externo ou recurso de *binding* para alguma linguagem, o OMNeT++, além de ser uma ferramenta proprietária, não atende as necessidades de usuários avançados que

gostariam de testar sua solução, aplicação ou controlador em um ambiente de grande escala como o NS-3, que pode chegar, com qualidade, entre 100 e 200 (NIKOLAEV, 2013) processos executando uma simulação.

Capítulo 3

IMPLEMENTAÇÃO E RESULTADOS

Este capítulo compreende toda arquitetura e implementação utilizados para chegar aos resultados a serem expostos na sequência. O primeiro ponto a ser considerado, ao pensar nas diferenças entre as formas (Simulação, Emulação e Método Híbrido), são as possíveis perdas e limitações no uso de dados, dado que um dos principais fatores da pesquisa na área de Redes e Sistemas Distribuídos é a entrega de dados, perdas, delay - dentre vários outros aspectos que, nesse contexto, podem variar os resultados, como já discutido em seções anteriores. A emulação, por utilizar dos recursos compartilhados entre os hosts e com diversas aplicações rodando no ambiente emulado, pode ter perda de desempenho, visto que estes ambientes são finitos, além de ter uma parcela reservada para a utilização do próprio sistema operacional hospedeiro. Já a simulação tem a garantia de uma entrega uniforme dos resultados, porém com uma complexidade inerente ao seu uso e com muitos recursos ainda não totalmente implementados para simulações, e por fim, uma forma híbrida do uso de ambos, combinando recursos emulados e simulados em um mesmo *layout* para entrega dos resultados. Esse método híbrido (ainda não abordado), pode ser utilizado de diversas formas. Alguns simuladores têm esse recurso pronto, justamente para facilitar a entrada de recursos do sistema hospedeiro, sem a necessidade de modelagem complexa presente na simulação. As trocas de mensagens entre ambiente emulado e simulado são tratadas uma a uma, visto que não existem dados reais sendo transmitidos na simulação e, para que os recursos emulados possam interagir da forma correta, esses dados têm de existir e terem contexto na aplicação ou sistema. Espera-se que a troca entre os ambientes (emulado e simulado) seja feita de forma transparente e sem um custo muito alto para o pesquisador.

3.0.1 Ambiente e Hardware

Para que todos os resultados tenham o mesmo ambiente, foi utilizada virtualização com VirtualBox, pela facilidade e isolamento fornecido pela tecnologia. A máquina hospedeira possui:

- Processador: Core i7-2670QM 2.2Ghz (4 núcleos 8 threads);
- Memória RAM: 16 Gigabytes DDR3 1600MHz;
- HD: SSD Kingston 256 Gigabytes;
- Suporte a virtualização: Intel VT-x/VT-x(EPS);
- Sistema operacional: Windows 10 - Home edition;

Existem três ambientes a serem preparados separadamente: emulado com o uso de Mininet; simulado com NS-3 e híbrido com Ns-3 e *containers* para emulação. Sendo assim, foram criadas três máquinas virtuais, todas com as mesmas reservas de hardware:

- Processador: 4 núcleos 8 threads;
- Memória RAM: 4 Gigabytes;
- HD: 25 Gigabytes;
- Sistema operacional: Xubuntu Desktop 64 bits (14.04 LTS);

Os requisitos acima selecionados para as máquinas virtuais superam os indicados e disponibilizados pelo site oficial do Mininet (LANTZ; HELLER; MCKEOWN, 2010) em seu *walkthrough* (com apenas 2 Gigabytes de memória e 2 núcleos reservados). Espera-se que as aplicações tenham processamento suficiente ou superior, com máxima performance e livre de gargalos. Artigos que também fizeram testes de performance com Mininet (KULDEEP K. SHARMA, 2014)(WANG, 2014) utilizaram recursos inferiores, permeando os 2 Gigabytes de memória e 1 ou 2 núcleos reservados, todos também virtualizados pelo VirtualBox. Com o aumento de memória e processamento disponível, espera-se atingir o máximo de banda possível. Tirando os pacotes específicos para a instalação de cada um dos ambientes, pacotes essenciais como *Python, CMAKE, GCC, GIT*, dentre outros, foram instalados e atualizados. Tendo essa base, a máquina foi replicada, destinando uma para cada ambiente, como já mencionado. É importante frisar que nenhuma máquina ou teste foi utilizado em paralelo com outro, visando garantir disponibilidade de recursos e desempenho para a execução.

3.0.2 Layout para base comum de dados

Para criar uma base de comparação entre os resultados atingidos com a implementação proposta e as capacidades comuns de cada ambiente, um layout simples de estrela foi desenvolvido com as mesmas características nos três ambientes:

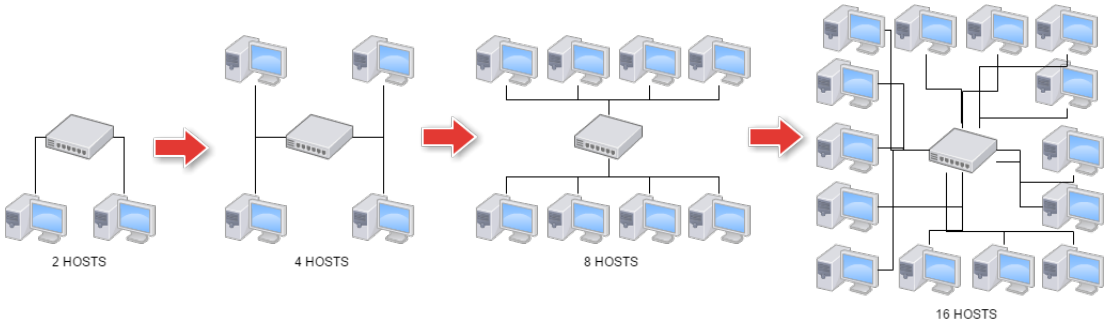

Figura 3.1: Primeira topologia e evolução para teste de largura de banda e entrega.[Própria]

Apesar de existirem diferenças contundentes na implementação de cada um (emulado, simulado e híbrido), os aspectos teóricos se aplicam da mesma forma entre os três. A figura 3.1 incia-se com apenas dois hosts interligados por um switch. O switch foi inserido para interligar os hosts, visto que existem mensagens de *broadcast* e um tempo de convergência, até que a rede possa comunicar normalmente. A figura sequencial demonstra o desenvolvimento teórico para os ambientes, cada qual sua particularidade, mas sempre respeitando a ideia de pares de transmissão (um emissor e um receptor) de forma mais randômica possível:

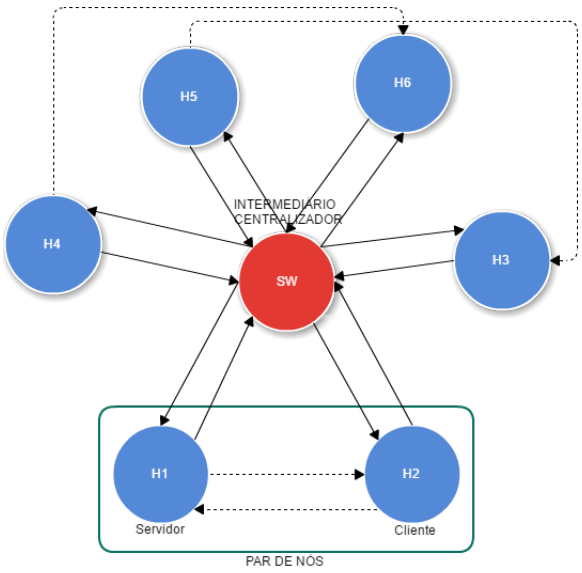

Figura 3.2: Esquema teórico para desenvolvimento das topologias.[Própria]

Todas as transmissões devem, de alguma forma, em sua tecnologia, serem capazes de coletar dados básicos para cálculos, como tamanho da janela e latência. Sendo o cálculo para

throughput:

```
TH = BandaMaximaDoNóReceptor / \textit{RoundTripTime}
```

A figura 3.2 começa em um único par, transmitindo até atingir 8 pares de transmissão simultânea, como demonstra a figura 3.1. Cada rodada acontece individualmente, em ambientes separados e não automatizados, ou seja, a execução não se dá automaticamente, começando do primeiro par até alcançar a última instância, com 8 pares. Cada uma deve ser criada e seus testes feitos individualmente, almejando a preservação de recursos da máquina virtual para melhor entrega de resultados e ruídos. Antes da execução de qualquer teste que visa coletar dados, o ambiente deve ser testado individualmente com testes de latência (*ping*) para validar o funcionamento correto da comunicação, antes mesmo de qualquer teste que visa coletar dados para o trabalho.

3.0.2.1 NS-3

Dando início aos trabalhos com o ambiente simulado, a versão do NS-3 escolhida foi a 3.26, sendo essa a penúltima versão e, portanto, a mais estável para instalação dos pacotes requeridos, o que se deu via documentação entregue pelo próprio ambiente. O NS-3 possui diversas formas de instalação de módulos separados mantidos pela comunidade NS-3. Foi escolhida, portanto, a versão "*full*"com todos os recursos disponíveis, inclusive o OpenFlow, já setando o ambiente para os próximos testes, conforme recomenda o *tutorial* (NS-3..., a). Utilizando de ferramentas disponíveis para NS-3, a instalação é, em sua maioria, automatizada via Mercurial (NS-3..., a) e Bake (BAKE...,). A instalação dos pacotes base necessários, no entanto, deve ser feita pelo próprio experimentador, já que as ferramentas de instalação apenas apontam os pacotes necessários e a falta de alguns deles, como segue:

```
> Python - OK
> GNU C++ compiler - OK
> Mercurial - OK
...
```

Essa instalação entrega todos os recursos básicos, além de módulos específicos, tais como OpenFlow(NS-3..., b), *click router*, CCNx e NS-3 *Simulation Cradle*(NS-3..., a).

Com o ambiente pronto, testes básicos com exemplos foram utilizados para certificar o funcionamento, o *tutorial* (NS-3..., a) aponta esses testes como necessários. Existe um *script* de teste próprio do NS-3 que entrega todos os testes de funcionamento:

```
./test.py

PASS: TestSuite netlink-socket
PASS: TestSuite process-manager
...
8 of 8 tests passed (8 passed, 0 skipped, 0 failed, 0 crashed, 0 valgrind errors)
```

Para teste básico de funcionamento e validação do ambiente, um *ping* entre máquinas apenas para determinar se as conexões e comunicação entre os nós se dá de forma limpa e sem ruídos. A execução do *ping* em um *layout* comum entre dois nós hosts é feita utilizando um dos exemplos prontos entregues na instalação do ambiente NS-3,

```
./waf build --enable-examples
```

com as modificações básicas para atender o desejado. Os resultados podem ser vistos ao habilitar o *trace*, já abordado durante a fundamentação teórica no tópico de ”Rastreamento dos Dados”, dois arquivos pcap relacionados a cada um dos nós são entregues. Isso se dá através da adição do seguinte trecho de código à simulação:

```
csma.EnablePcapAll ("csma-ping", false);
```

Desta forma, temos o csma como objeto responsável por gerar todos os links de comunicação. O método ”*EnablePcapAll*”permite a coleta de forma individual do meio e dos nós. Assim sendo, a análise do teste de funcionamento do ambiente se dá em três etapas:

- Execução: o código executou sem nenhum problema;
- Finalização: nenhum erro inesperado foi entregue ao final ou durante a execução da simulação;
- Resultados: foram entregues arquivos, saídas ou qualquer nuance de *log* da execução de forma compreensível e esperada.

A execução e finalização não retornaram erros e os resultados entregaram dois arquivos pcap correspondentes a cada um dos hosts da simulação inicial de testes.

Vale ressaltar que, como apontado no capítulo de Trabalhos Relacionados, a existência do DCE (*Direct Code Execution*)(TAZAKI, 2013) tem uma instalação separada por utilizar de

recursos específicos do sistema operacional e não faz parte desse pacote de simulação "*all in one*".

Para atender aos requisitos do teste referenciado pelas figuras 3.1 e 3.2, foram utilizadas classes presentes no NS-3 para testes, utilizando o protocolo UDP:

- Gerador de tráfego: udp-client.h (gerador de tráfego) controlado pela classe onoffapplication.h (responsável por criar sequências de transmissão);
- Recepção de dados: udp-server.h (preparado para receber os dados);
- Coleta de informações: flowmonitor (completo tratamento de dados)(CARNEIRO; FORTUNA; RICARDO, 2009);

A justificativa do uso de UDP nesse primeiro teste é poder atingir o máximo possível e disponível em banda. Como serão feitos testes com diferentes tipos de links (100MB, 1G, 10G), é interessante que o gerador de tráfego atinja o máximo possível em um determinado link disponível. Para os testes com TCP, outras classes também presentes foram utilizadas:

- BulkSenderHelper: gera os dados necessários para a estimativa de banda, é possível definir a quantidade de dados a serem enviados;
- PacketSinkHelper: visto que os dados gerados são em TCP, é necessário que haja um receptor para esses dados. A classe PacketSink tem essa funcionalidade de recepção de dados TCP;
- Coleta de informações: flowmonitor (completo tratamento de dados)(CARNEIRO; FORTUNA; RICARDO, 2009);

O *BulkSendApplication*(NS-3..., a) utiliza o máximo de banda possível para testes (bem como o *iperf*), utilizando o protocolo TCP e gerando resultados suficientes para comparações com o *iperf* (havendo a opção comum em TCP ou gerando dados diretamente com UDP). A diferença entre a abordagem utilizando TCP e UDP são os métodos entre cliente e servidor: enquanto para o uso de UDP existe uma aplicação cliente/servidor, o TCP necessita de outra classe para sincronia e envio dos dados. Os exemplos a seguir demonstram trechos de código referentes à geração de tráfego e se repetem em todos os testes:

```
//Aplicação servidor
uint16_t port = 4000;
```

```
UdpServerHelper server (port);
ApplicationContainer apps = server.Install (n.Get (1));

//Aplicação cliente
UdpClientHelper client (serverAddress, port);
client.SetAttribute ("MaxPackets", UintegerValue (maxPacketCount));
client.SetAttribute ("Interval", TimeValue (interPacketInterval));
client.SetAttribute ("PacketSize", UintegerValue (MaxPacketSize));
apps = client.Install (n.Get (0));
```

A aplicação UDP pode ser customizada ao ponto de escolha no número de pacotes a serem transmitidos, intervalo entre os pacotes e tamanho do pacote. Ainda é possível observar que existe uma escolha de porta para que a aplicação se comunique dentro do ambiente simulado. Já para a utilização dos testes em TCP, foram alterados para a aplicação acima mencionada do *BulkSend* e *PacketSink*:

```
uint16_t port = 9;
BulkSendHelper source ("ns3::TcpSocketFactory", InetSocketAddress (i.GetAddress (1)
source.SetAttribute ("MaxBytes", UintegerValue (maxBytes));
ApplicationContainer sourceApps = source.Install (nodes.Get (0));

PacketSinkHelper sink ("ns3::TcpSocketFactory",
InetSocketAddress (Ipv4Address::GetAny (), port));
ApplicationContainer sinkApps = sink.Install (nodes.Get (1));
```

Dando fim as especificações que permeiam o uso de simulação nessa implementação, ainda é importante lembrar do uso do FlowMonitor, sendo que as estatísticas apresentadas nos resultados são diretamente ligadas ao seu uso, ou seja, por mais que as aplicações entreguem seus dados em específico, sempre serão utilizadas as informações dispostas pelo FlowMonitor em XML(CARNEIRO; FORTUNA; RICARDO, 2009), como será abordado em breve na comparação entre a entrega do *iperf* e *flowMonitor*.

O código de referências para todas as aplicações encontra-se no Apêndice A.

3.0.2.2 Mininet

Bem mais prático e polido em termos de instalação e utilização que o simulador NS-3 o Mininet (já devidamente apresentado anteriormente)(MININET..., a), faz parte dos principais

repositórios e a instalação pode ser feita a partir de uma simples linha de comando:

```
apt-get install mininet
```

ou

```
yum install mininet
```

Diferente do NS-3, que tem diversos pacotes e recursos necessários que devem ser instalados separadamente, o Mininet já adiciona todos seus recursos, inclusive o OVS (app switch que para a ligação entre os *containers*), como demonstrado na figura 2.5. Todas as necessidades para um teste básico de funcionamento, como o proposto, não necessitam modelar o ambiente. A CLI rica do Mininet permite todo teste necessário, similar ao feito no ambiente NS-3:

```
mn --test pingall --topo single,2
```

O resultado, devido ao tratamento inerente ao Mininet, apresenta a comunicação com sucesso, diretamente na CLI própria:

```
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

Figura 3.3: Teste de comunicação básica antes dos experimentos.

Pelas classes presentes e já discutidas anteriormente, para o teste de funcionamento do ambiente, o resultado anterior é satisfatório: executando, finalizando e com resultados.

O teste de *throughput* foi feito utilizando o *iperf*, presente no próprio Mininet. A validade dos resultados obtidos entre diferentes ferramentas será abordada em uma próxima seção, demonstrando a justificativa que compare os dados de forma fidedigna. Para o momento, assim como no *layout*, em NS-3 os testes se iniciaram com apenas dois hosts, para o teste de banda com OpenFlow em uma aplicação de switch comum. A particularidade de aproximação utilizada com o Mininet é a especificação direta do uso do OVS como switch para interconexão. Apesar de o OVS ser inerente ao Mininet, por questões de transparência no código, o mesmo foi especificado. A implementação base dos testes com Mininet encontra-se no Apêndice B.

3.0.2.3 Híbrido

Nem todo pesquisador tem facilidade em modelar o objeto de pesquisa de forma que possa participar de uma simulação. Nesse contexto, é interessante que objetos ou aplicações do

mundo real possam se comunicar com o mundo simulado, exatamente como funciona o EstiNet(TECHNOLOGIES,) e a inserção de aplicações pelo DCE (**??**). Existem classes específicas no NS-3 que, combinadas, podem se comunicar através de uma interface emulada e do uso de túneis. O processo de instalação não difere do NS-3, com a exceção de que é necessário instalar os recursos para *container* LXC. A imagem demonstra como a comunicação se dá:

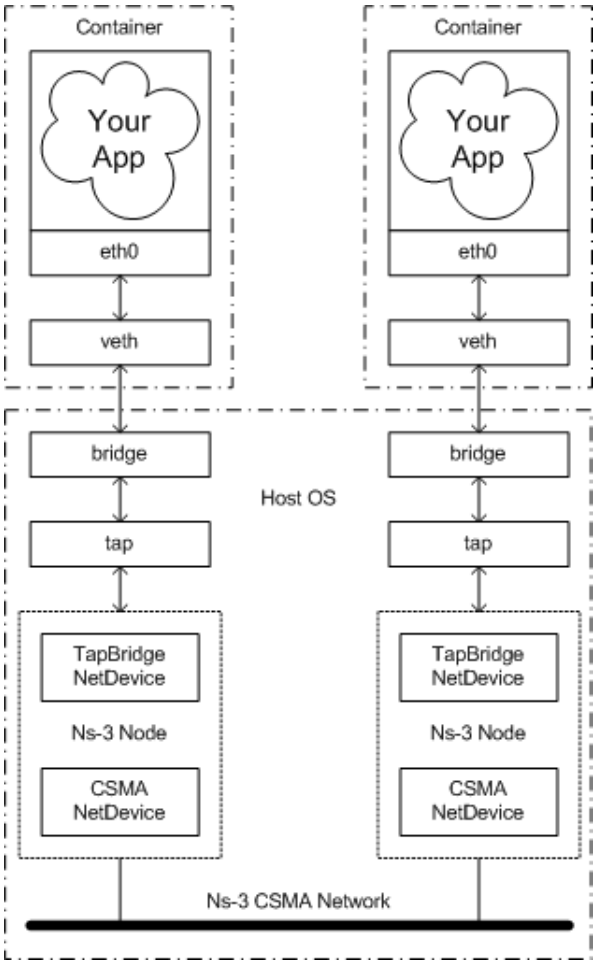

Figura 3.4: Comunicação entre ambiente simulado e emulado com NS-3 e LXC (NS-3..., a).

Nesse ambiente, temos três partes: *namespace* do usuário (*userspace*), *namespace* do NS-3 e *namespace* dos *containers* para cada nó. Para exemplificar melhor, vamos seguir o fluxo de um teste *ping* comum: a aplicação rodando em um *container* cria o *echo request*, passando para a interface virtual, conectada a uma *bridge* sem nenhum outro *container* dividindo. A *bridge*, portanto, está conectada a um túnel TLS diretamente ligado ao ambiente simulado NS-3 (classe específica para uso de TLS). Já no ambiente simulado, o pacote é tratado como um objeto e seus eventos ocorrem de forma normal, ou seja, o pacote é apenas representativo na simulação, mas corre por toda ela até chegar ao outro lado e fazer o processo inverso. Esse processo garante que exista comunicação de forma confiável entre os ambientes, porém não existe nenhum *helper* (classe especial que auxilia no uso de métodos em NS-3) que possa simplificar a criação

desse ambiente com *containers*, em LXC. Outro ponto é que não existe um controle remoto entre o ambiente simulado e o emulado para poder automatizar testes. Dessa forma, qualquer necessidade de comunicação entre os *containers* só pode ser feita de forma manual. O primeiro teste de comunicação segue o tutorial de LXC com NS-3 (NS-3..., a). É um *ping* entre dois *containers*, criados a partir do *template* ubuntu. O processo completo pode ser verificado no apêndice B. Um último ponto importante a ser ressaltado é a necessidade de a execução da simulação deve ter um tempo alto, para que exista tempo suficiente do experimentador utilizar os *containers*.

Seguindo o praxe dos testes de ambientes anteriores, foi feito um *ping* entre hosts, para validar o ambiente com o seguinte *layout*:

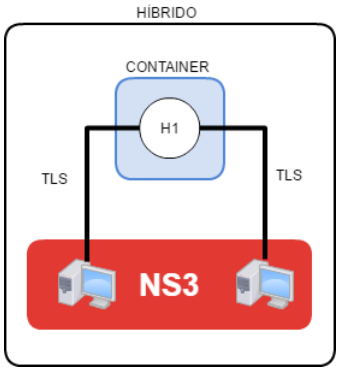

Figura 3.5: Topologia híbrida.

Observando a figura 3.5, é possível notar que a atuação dos geradores de tráfego é inversa. Os hosts estão em ambiente simulado e a área emulada recebe o *container* com OVS. Vale ressaltar que os hosts em ambiente emulado não possuem comunicação entre eles sem passar pela área emulada. Nesse aspecto, é importante relembrar a arquitetura da figura 3.4, em que a comunicação simplificada pela figura 3.5 possui *bridges*, túneis e a interface virtual criando *overhead* na transmissão entre os pontos.

Os resultados a seguir demonstram os dados sumarizados da implementação. O primeiro ponto de teste foi representado pela figura 3.1. É demonstrado, por meio de tais resultados, um teste de uso de banda com o aumento de hosts, trabalhando em conjunto e de forma randômica para os pares cliente/servidor:

Existe uma queda ainda maior de banda quando é utilizado o protocolo TCP, visto que é inerente o uso de confirmação. O uso de simulação manteve-se fiel ao link e a banda. Todos os procedimentos de geração de pacote estão descritos anteriormente.

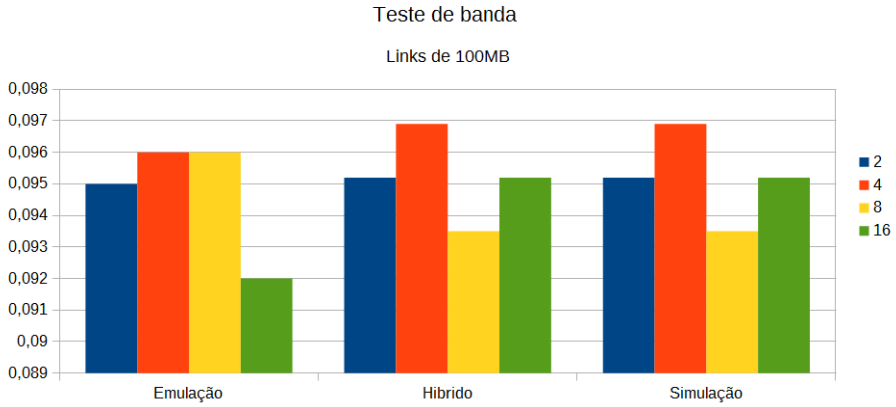

Figura 3.6: Teste de banda com links setados a 100MB, delay 1ms.

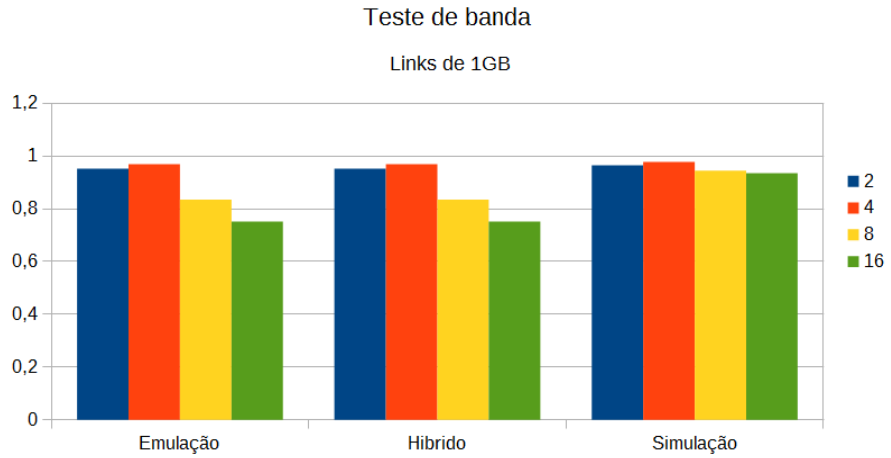

Figura 3.7: Teste de banda com links setados a 1GB, delay 1ms.

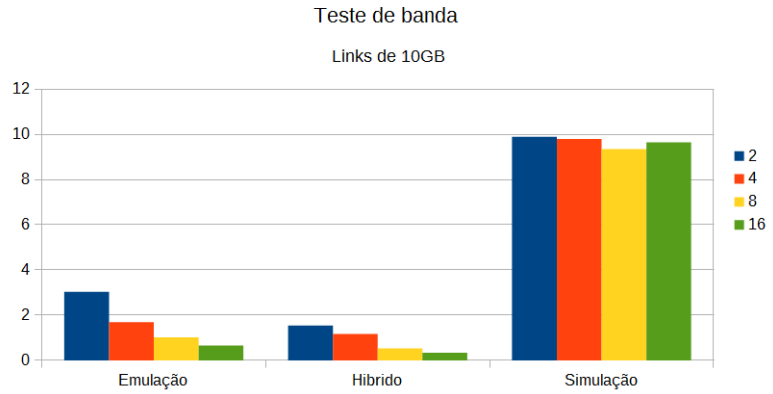

Figura 3.8: Teste de banda com links setados a 10GB, delay 1ms em UDP.

3.0.3 Comparação de tracing

Como mencionado anteriormente, os resultados entre os ambientes foram coletados de forma diferenciada, justamente por não existir uma ferramenta de análise universal. O método

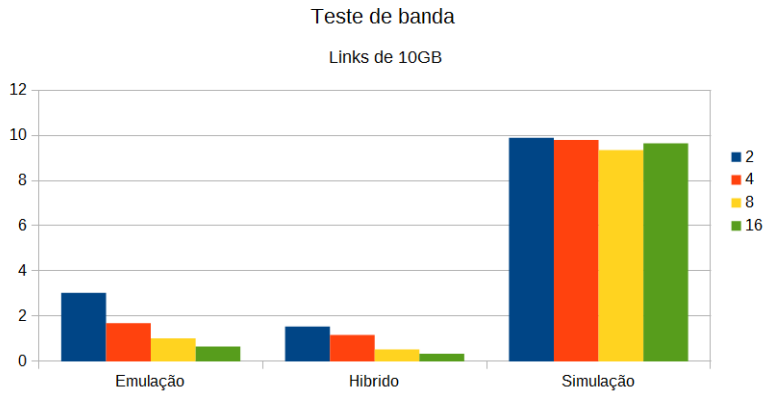

Figura 3.9: Teste de banda com links setados a 10GB, delay 1ms em TCP.

para coleta de dados com o ambiente simulado foi o *FlowMonitor*. Para o Mininet, foi escolhido o uso da ferramenta *iperf*. Nesse momento, é possível indagar a qualidade dos resultados, visto que são ferramentas e e = repetido, tirar formas diferentes de coleta de dados. Felizmente, o *FlowMonitor* e o *iperf* entregam os mesmos dados, cada qual com sua plataforma. A figura a seguir demonstra todos os cálculos realizados pelo *Flow Monitor*:

mean delay: $\overline{delay} = \frac{delaySum}{rxPackets}$

mean jitter: $\overline{jitter} = \frac{jitterSum}{rxPackets-1}$

mean transmitted packet size (byte): $\overline{S_{tx}} = \frac{txBytes}{txPackets}$

mean received packet size (byte): $\overline{S_{rx}} = \frac{rxBytes}{rxPackets}$

mean transmitted bitrate (bit/s):

$$\overline{B_{tx}} = \frac{8 \cdot txBytes}{timeLastTxPacket - timeFirstTxPacket}$$

mean received bitrate (bit/s):

$$\overline{B_{rx}} = \frac{8 \cdot rxBytes}{timeLastRxPacket - timeFirstRxPacket}$$

mean hop count: $\overline{hopcount} = 1 + \frac{timesForwarded}{rxPackets}$

packet loss ratio: $q = \frac{lostPackets}{rxPackets+lostPackets}$

Figura 3.10: Cálculos realizados pelo *FlowMonitor*.(CARNEIRO; FORTUNA; RICARDO, 2009)

É possível observar que os cálculos realizados pelo *iperf* também se encontram na entrega do *FlowMonitor*, igualando a entrega de resultados entre ambos.

Todos os dados são entregues em um arquivo XML, o qual concentra todas as informações dos cálculos acima descritos:

```
<FlowStats>
<Flow flowId="1" timeFirstTxPacket="+0.0ns" timeFirstRxPacket="+20067198.0ns" timeL
</Flow>
```

3.0.4 Consumo de recursos e desempenho

Cada ambiente de testes costuma consumir uma determinada quantidade de recursos de *hardware* para entrega dos resultados, ou seja, no caso de simulação, emulação, híbrido ou mesmo uma *testbed*, as limitações de máquina podem interferir ou mesmo criar flutuações nos dados finais a serem entregues, visto que, independente do método, todos consomem processamento, memória e até mesmo disco. Desta forma, é importante apontar o consumo para cada uma das execuções, de forma a ver a influência nos resultados ou mesmo a necessidade de cada *layout*.

Os testes de consumo se baseiam na execução do *layout* com 2 e 16 hosts e a coleta dos dados se dá através de métricas entregues diretamente pelo VirtualBox. Para esclarecer a coleta das métricas, as próximas linhas darão uma explicação direta de coleta das métricas e *setup* do mesmo. O VirtualBox disponibiliza uma suíte de aplicativos que podem, não apenas controlar aspectos de configuração (reserva de memória, realocação de disco, número de núcleos de processamento por máquina - dentre outros), como também métricas que podem ser configuradas individualmente (VBOXMANAGE,). Executado diretamente via linha de comando do computador hospedeiro, o VBoxManager permite (VBOXMANAGE,):

- Utilização de uma GUI diferente da padrão do VirtualBox. Existem algumas interfaces diferenciadas que só podem ser habilitadas através da CLI;
- Acesso a recursos experimentais e opções avançadas não disponíveis na GUI padrão;
- Todas as opções padrão da GUI;
- Capacidade de interagir e alterar discos e características da máquina virtual.

Uma das opções a ser melhor explorada são as métricas, que podem ser entregues por uma máquina virtual ou domínio específico. Como as métricas são várias e diversificadas, serão concentradas apenas as utilizadas e alguns exemplos. Todos os comandos devem de ser executados diretamente do prompt de comando (seja Windows ou Linux) por meio do diretório de instalação padrão do VirtualBox. Por tabela, todas as métricas disponíveis são coletadas da máquina virtual, se não forem especificadas quais métricas devem ser retornadas. Todas as opções disponíveis encontram-se no comando,

```
VboxManage metrics list
```

que disponibiliza (dentre várias outras) as seguintes métricas:

- CPU/Load/User: status de consumo de processamento do computador hospedeiro;
- CPU/Load/Iddle: tempo sem uso de processamento;
- CPU/MHz: clock do processador do hospedeiro (variável em computadores com *turbo-clock*);
- RAM/Usage/Used: quantidade de memória utilizada;
- RAM/Usage/Free: quantidade de memória livre;
- Disk/Usage/Used: total de uso do disco (virtual).

Para cada uma das opções (sem contar as métricas específicas que forem calculadas pelo computador hospedeiro), podem ser especificados diversos outros dados a serem coletados, tanto para o hospedeiro quanto para a máquina virtual em que se deseja coletar, lembrando que a coleta se dá de forma individual. Escolhendo as métricas é necessário criar o *setup*, como segue:

```
VBoxManage metrics setup --period 1 --samples 5 NOME-VM CPU/Load:avg,RAM/Used
```

O comando acima determina que serão coletadas métricas a cada ciclo, com 5 amostras da VM com o nome determinado em NOME-VM e em conjunto com quais das métricas (de todas as disponíveis) serão coletadas e de que forma. Para cada uma, é possível ainda determinar se será coletado o valor integral, o menor valor (mínimo), maior valor (máximo) ou a média (avg). No exemplo acima, são coletadas as médias de uso de processamento e memória, sendo que, para coleta em tempo real, é necessário um comando específico. O exemplo a seguir demonstra o comando e a saída do *setup*:

```
VBoxManage metrics collect

Linux-NS3 CPU/Load/User:avg     5.8%
Linux-NS3 RAM/Usage/Used        109420 kB
```

O comando roda até que seja parado manualmente. As métricas sempre continuam estáticas, até que sejam substituídas por novas, ou seja, o *setup* se mantém. Para termos uma base de uso de memória para o sistema e processamento, foram coletadas algumas rodadas de experimentação, sem a execução dos *layouts*:

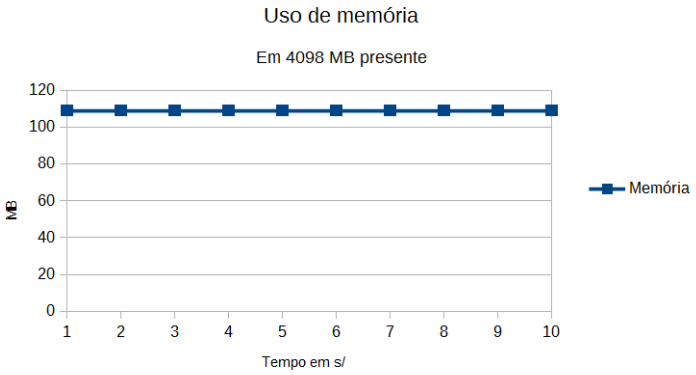

Figura 3.11: Consumo de memória sem nenhum experimento.

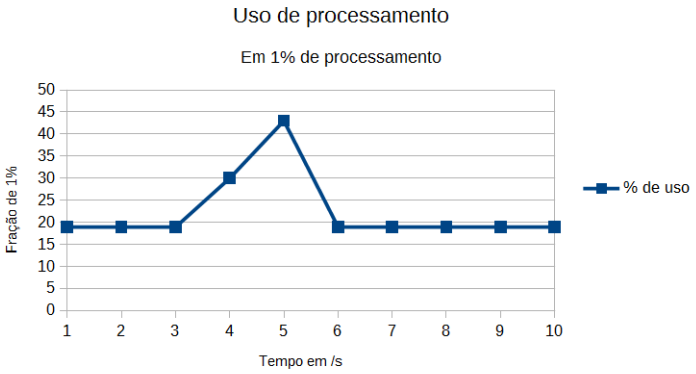

Figura 3.12: Consumo de processamento sem nenhum experimento.

Uso de CPU e memória mantém uma média, sem muita variação dos dados coletados na máquina virtual sem uso de nenhum programa ou execução de algum processo. É possível observar que a variação não ultrapassa o 1% de uso de processamento e aproximadamente 110MB de memória. O gráfico a seguir mostra as variações entre cada um dos ambientes nas execuções de 2 e 16 hosts em uso de processamento e memória, respectivamente:

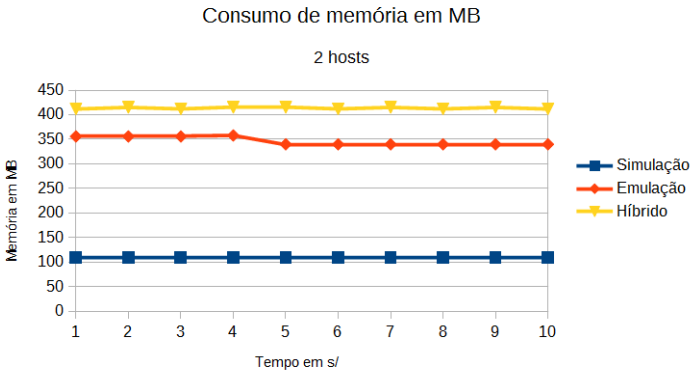

Figura 3.13: Consumo de memória para 2 hosts em execução do experimento.

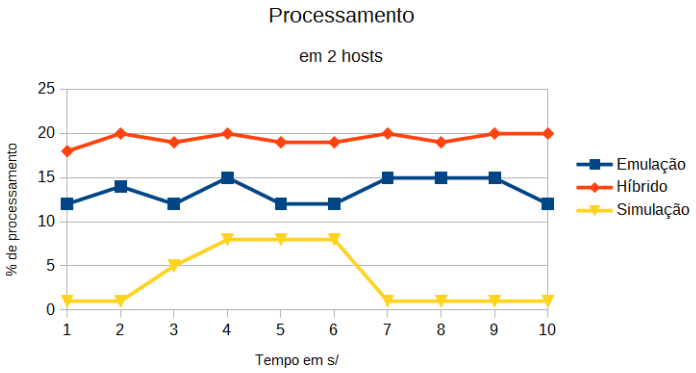

Figura 3.14: Consumo de processamento para 2 hosts em execução do experimento.

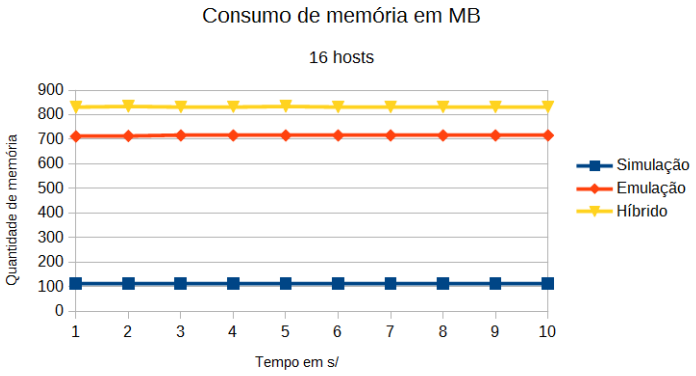

Figura 3.15: Consumo de memória para 16 hosts em execução do experimento.

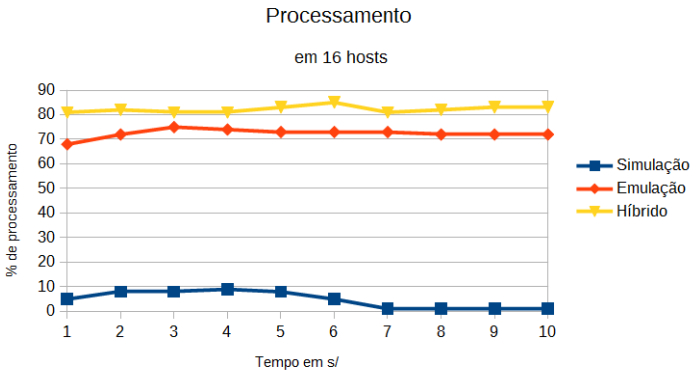

Figura 3.16: Consumo de processamento para 16 hosts em execução do experimento.

As variações são mais fortes e presentes nos ambientes emulados, enquanto a variação no ambiente simulado não ultrapassa os 10%, bem como uso de memória, que se mantém. É um comportamento previsível, dado que tanto o Mininet quanto o método híbrido necessitam de recursos da máquina, por meio do uso de *containers*. Ainda deve-se considerar que, para o ambiente híbrido, são usadas *bridges* e túneis para a comunicação entre o ambiente simulado e o emulado. Este consumo crescente pode ser problemático, de acordo com o número de nós, dependendo do *layout* desejado. A emulação de ambientes muito complexos, como proposto anteriormente, pode sofrer um grande ruído nos resultados entregues pelo *hardware*. Apesar do consumo elevado, os resultados ainda se mantém estáticos, mesmo com uma grande quantidade de memória e alguma quantidade de processamento sobrando. A multitarefa entre os *containers* pode ser um dos fatores que influenciam nesse resultado. É importante observar que o consumo se mantém, de certa forma, estático, consumindo tudo que for necessário em um primeiro momento sem muitas flutuações durante a execução.

3.0.5 STP OpenFlow

A aplicação OpenFlow escolhida para o *tradeoff* entre os ambientes foi uma implementação do protocolo *Spanning Tree*, por alguns motivos:

- Diversos ambientes *datacenter* utilizam topologias fortemente interligadas, tornando o protocolo necessário;
- O NS-3 ainda não possui uma forma de utilizar diversos switches interconectados;
- O *loop* de links é presente em diversos usos de diversas tecnologias, sendo a topologia linear a mais presente no NS-3.

Conseguimos criar, portanto, a ideia de que o uso do STP com OpenFlow pode ser utilizado em locais em que o uso de Redes Definidas por Software já são comuns. De forma bem simplificada, o algoritmo STP em Redes Definidas por Software(POX...,) busca todos os links que se repetem entre ativos OpenFlow que começam a fazer parte da rede. Faz parte da tecnologia de Redes Definidas por Software a visão completa da rede, como já abordado na Fundamentação Teórica. Experimentos com *layout* de *datacenter* são cada vez mais comuns em pesquisas, fazendo do uso do protocolo uma necessidade básica e inerente. Para os ambientes emulados e híbrido, a escolha foi do controlador em Pox, utilizando uma implementação STP, e para o ambiente simulado, foi escolhida/utilizada uma modificação no controlador padrão já disponível , como será explicado em cada uma dos tópicos a seguir.

3.0.6 Soluções nos ambientes

Os testes com STP devem, obrigatoriamente, conter uma topologia que faça sentido do ponto de vista da correta utilização do protocolo. Uma proposta interessante é apresentada pelo artigo (AL-FARES; LOUKISSAS; VAHDAT, 2008), o qual propõe a seguinte topologia:

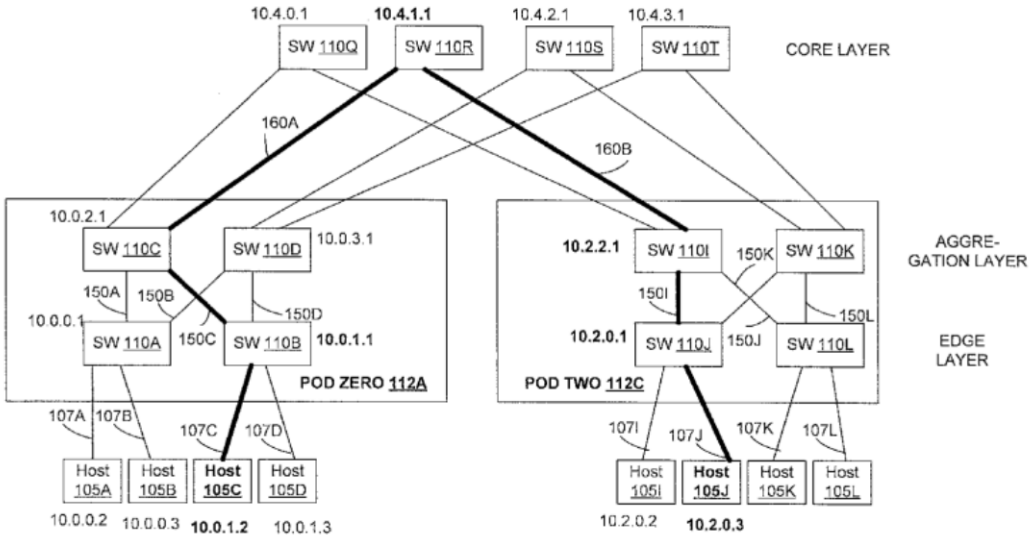

Figura 3.17: Topologia datacenter. (AL-FARES; LOUKISSAS; VAHDAT, 2008)

Para os testes dos ambientes, foi escolhido um *pod* da topologia da figura 3.17, que apresenta características suficientes para criar o *layout* com os testes STP feitos com OpenFlow. Baseado na figura acima, foi construído o seguinte *layout*:

Apenas reforçando, essa topologia será construída para os três ambientes que serão discutidos e abordados separadamente, com suas particularidades no desenvolvimento.

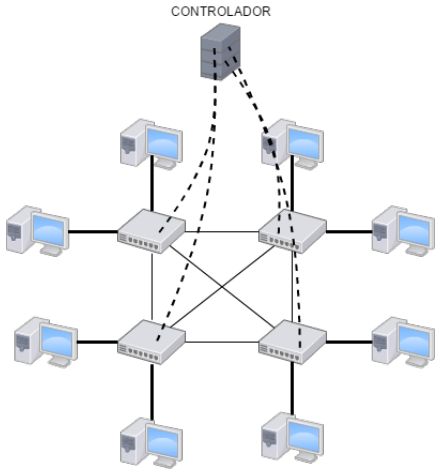

Figura 3.18: Topologia Fat Tree com OpenFlow e controlador. [Própria]

3.0.6.1 Mininet

Utilizando-se da API disponível para desenvolvimento e diferente do *layout* anterior, todos os hosts já foram inseridos e os testes foram gradativos por meio da utilização do arquivo de entrada disponível em um dos construtores do Mininet. Existe um tempo de convergência claro e necessário para que o ambiente esteja disponível, característica do uso de OpenFlow. Não há um protocolo STP rodando, e sim um controle das portas via controlador, o que diminui a quantidade de *broadcasts* na rede, antes que ela esteja totalmente livre para os testes - bem como no exemplo de *throughput* com diversos dispositivos, em que o arquivo externo que = tirar, determinou os testes e a sequência em que seriam feitos. Escrito para uso da API em Mininet e utilizando-se das classes já presentes no código, os switches foram adicionados especificadamente em (?) OVS, com controlador externo e uma série de testes de acordo com o arquivo apontado, como segue:

```
net = Mininet( topo=None, build=False, "tests.txt" )
...
net.addController('c0', controller=RemoteController,ip="127.0.0.1",port=6653)
...
s1 = net.addSwitch( 's1', cls=OVSSwitch )
s2 = net.addSwitch( 's2', cls=OVSSwitch )
s3 = net.addSwitch( 's3', cls=OVSSwitch )
s4 = net.addSwitch( 's4', cls=OVSSwitch )
...
net.pingall()
CLI( net )
```

...

O comando *net.pingall()* ajuda a convergir a rede antes dos testes, para que não aconteça a perda de pacotes iniciais até a instalação de regras OpenFlow. Os switches são todos compostos por instâncias de OVS: como pode ser notado, o controlador roda no *userspace* e o arquivo *tests.txt* possui os testes de *iperf* para todos os nós.

Como de praxe, apesar da necessidade de compreensão da API, o desenvolvimento em Mininet toma bem menos tempo em qualquer contexto desejado.

3.0.6.2 NS-3

O ambiente mais complexo de desenvolvimento, dentre todos os apresentados, tem um um problema intrínseco com o uso de Redes Definidas por Software, pois não existe nenhuma forma de comunicação com controladores externos para OpenFlow. Outro ponto muito problemático, não existir protocolo STP nos switches modelados, atuando apenas como bridges compartilhadas entre as conexões. Esse tipo de problema, em uma modelagem de um ambiente como o proposto na figura 3.17, ou na figura 3.18, é impossível pelos broadcasts criados em qualquer comunicação de dados. A figura a seguir demonstra o problema ao utilizar loops em NS-3:

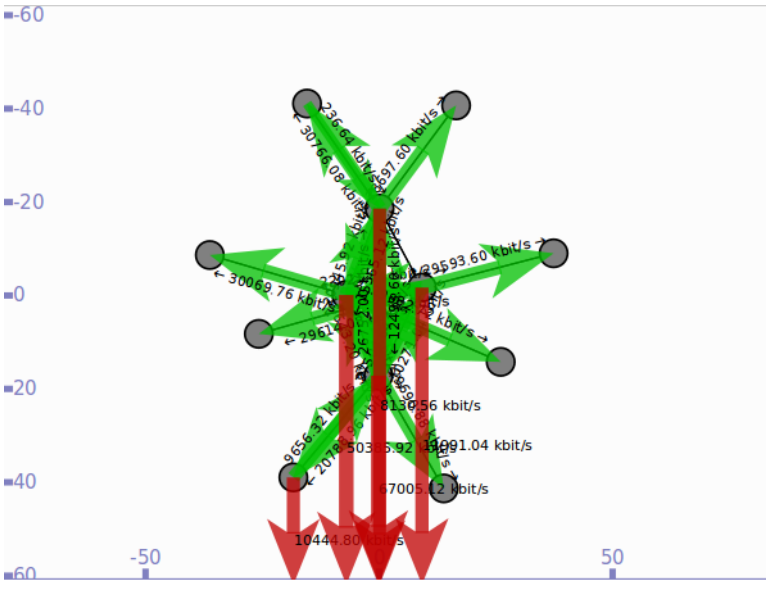

Figura 3.19: Resultado de uma simulação sem o uso de STP. [Própria]

Sendo assim, a simulação tem de ser encerrada manualmente, por nunca conseguir cumprir seu objetivo. Foi observado, em testes, que em cerca de uma hora de simulação, ainda não havia sido finalizado um simples ping entre hosts. Outro ponto que demonstra a elegibilidade do uso de STP é a seguinte mensagem, presente nos arquivos de modelagem do switch comum e Openflow:

```
* \attention The Spanning Tree Protocol part of 802.1D is not
* implemented.  Therefore, you have to be careful not to create
* bridging loops, or else the network will collapse.
```

Para que o ambiente referido pudesse ser utilizado com OpenFlow, o desenvolvimento teve que ser feito diretamente no módulo controlador disponível e funcional , já presente no NS-3. Isso já demonstra um dos problemas a serem solucionados. Todos os objetos compilados OpenFlow estão presentes nos NS-3 e encontram-se em um único local:

- ns-3.26/build/src/openflow -¿ possui os objetos já compilados;
- ns-3.26/src/openflow -¿ modelos do switch OpenFlow, OpenFlow net device, OpenFlow interface;
- ../openflow-ns-3.26/controller -¿ controlador OpenFlow.

Já foi explicado anteriormente o que são os helpers e outros pontos de modelagem, assunto este o qual não será abordado novamente. O arquivo de modelagem encontra-se no mesmo local da pasta de instalação, junto com vários outros. Iniciando pelo switch, existem métodos que foram essenciais para a implementação. O modelo possui todas as funcionalidades comuns presentes em um switch OpenFlow comum. Os métodos a seguir auxiliarão no entendimento da construção do controlador:

```
OpenFlowSwitchNetDevice::OutputPacket (uint32_t packet_uid, int out_port)
OpenFlowSwitchNetDevice::DoOutput (uint32_t packet_uid, int in_port, size_t max_len
OpenFlowSwitchNetDevice::FillPortDesc (ofi::Port p, ofp_phy_port *desc)
OpenFlowSwitchNetDevice::GetPorts ()
Ptr<ofi::Controller> m_controller.
```

Os métodos acima auxiliaram na ideia de blocking do controlador. Também existem outros métodos interessantes, como o de análise direta de broadcast, que pode ser impedido por um roteador ou terminal, ou o de forward, sem consulta da tabela de fluxos. Todos os pacotes trabalham com identificação, o que facilita a execução de um trabalho refinado, pacote a pacote. Apesar de nem todos os métodos presentes no switch criarem um *trace* de log para fácil identificação ou *debug*, é possível adicionar de forma fácil esses recursos em qualquer parte da simulação de forma específica. Uma interface OpenFlow tem, em sua estrutura, as seguintes linhas de código:

```
struct Port
{
Port () : config (0),
state (0),
netdev (0),
rx_packets (0),
tx_packets (0),
rx_bytes (0),
tx_bytes (0),
tx_dropped (0),
mpls_ttl0_dropped (0)
{
}
```

Portanto, é possível modificar o arquivo de modelagem, para que as portas que têm o loop tenham a flag OpenFlow de DROP, impedindo, dessa forma, o loop e criando a topologia linear sobreposta à FatTree.

Para a desativação das portas extras, foi adicionado mais um método presente no modelo de switch, de forma que ele pudesse desativar o NetDevice que possui o link em loop. O arquivo de modelo de switch Openflow foi alterado diretamente como segue:

```
OpenFlowSwitchNetDevice::UpdatePortStatus (ofi::Port& p)
{
uint32_t orig_config = p.config;
uint32_t orig_state = p.state;

p.config &= ~OFPPC_PORT_DOWN;

if (p.netdev->IsLinkUp ())
{
p.state &= ~OFPPS_LINK_DOWN;
}
else
{
p.state |= OFPPS_LINK_DOWN;
}
```

```
}

++ OpenFlowSwitchNetDevice::UpdatePortStatus (ofi::Port& p, bool state)
++ {
++  if (state == false){
++  p.state &= OFPPS_LINK_DOWN;
++  } else if (state == true ) {
++  p.state  &= ~OFPPS_LINK_DOWN;
++  }
++ }
```

É importante ressaltar que, na implementação OpenFlow do NS-3, existem tipos diferentes de interface, ou seja, o NetDevice e Interface são modificações dos módulos presentes, com a inserção de funções providas pela biblioteca externa e padrão do OpenFlow para ambientes simulados. O switch, portanto, não passa de uma bridge com interfaces OpenFlow, além do tratamento das flags e mensagens OF. O próximo passo foi trabalhar direto com o controlador. Já existem duas implementações presentes:

- Learning Switch: implementação comum de um switch qualquer;
- DropController: drop de todos os pacotes.

A forma mais fácil e evidente de adicionar algum controle STP é modificar o Learning Switch para ignorar mensagens de portas redundantes. Para facilitar a implementação desse algoritmo, foram consideradas as seguintes premissas:

1. As portas 1 e 2 sempre serão hosts;
2. As portas 3, 4 e 5 sempre serão os links redundantes para outros switches;
3. Apenas a porta 3 irá transmitir;
4. Portas 4 e 5 irão ignorar toda a comunicação.

Ignorar a comunicação e não simplesmente retirar o link, espera-se que, em implementações futuras, isso não seja feito de forma tão estática como o apresentado neste trabalho. Importante ressaltar que a entrega não é um protocolo STP OpenFlow dinâmico e totalmente funcional, mas

sim o *tradeoff* entre os ambientes. Com as premissas anteriormente numeradas, cada porta, em sua sequência, tem algumas *Flags* apontadas para *DROP*.

As modificações a seguir foram realizadas diretamente no código controlador:

```
new_switch(struct switch_ *sw, struct vconn *vconn, const char *name) {
sw->rconn = rconn_new_from_vconn(name, vconn);

++ ofPorts[] = sw.GetNSwitchPorts()
++ sw.UpdatePortStatus (ofPorts[3], bool false)
++ sw.UpdatePortStatus (ofPorts[4], bool false)

sw->mswitch = mpls_switch_create(sw->rconn);
sw->lswitch = lswitch_create(sw->rconn, learn_macs,
setup_flows ? max_idle : -1);

}
```

Dessa forma, a cada nova entrada de *switch*, é utilizado o ponteiro para desativar as demais portas redundantes, evitando, assim, o *loop* de mensagens.

3.0.6.3 Híbrido

Composto tanto por emulação, quanto por simulação, esse ambiente caracteriza-se por ser mais trabalhoso de se lidar e seu manejo é quase que totalmente manual. Para cada novo switch há um novo container, bridge e todos os outros demais recursos de sistema já apontados anteriormente. Para melhor exemplificar, a imagem a seguir demonstra como ficou o setup deste ambiente:

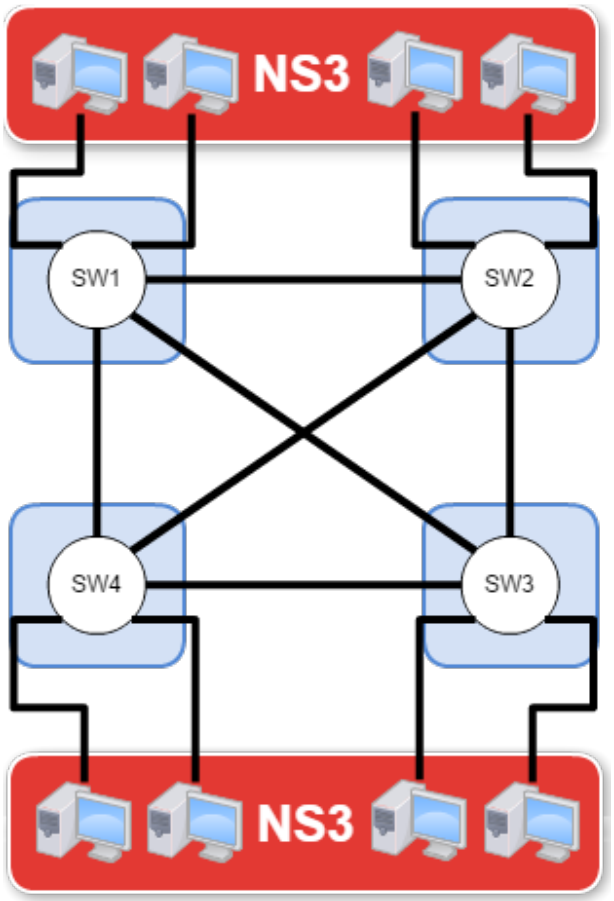

Figura 3.20: Layout final de método híbrido. [Própria]

Não existe nenhuma automação da execução dos códigos. Nesse ambiente, como os hosts existem apenas na simulação, os dados foram coletados também utilizando o FlowMonitor já discutido anteriormente. Vale ressaltar que todo caminho de cada dado passa por diversas *bridges* e túneis antes de chegar em qualquer uma das pontas de comunicação, aumentando significantemente o *overhead*.

A figura a seguir representa os testes, com a adição dos *switches* e a solução de comparação OpenFlow:

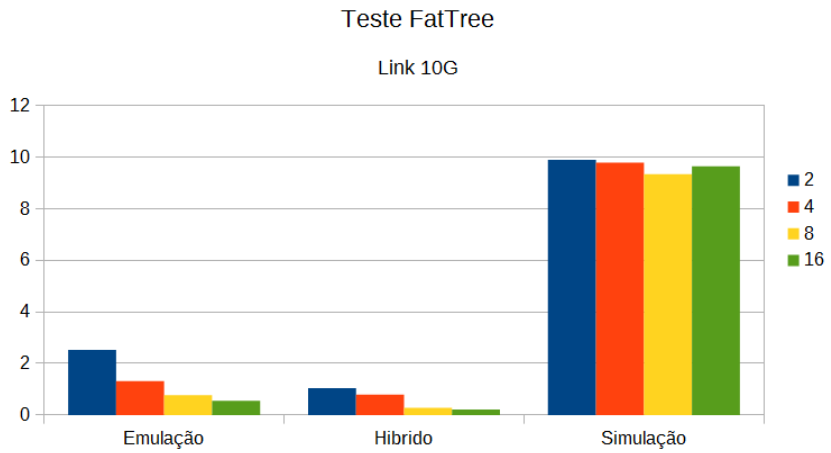

Figura 3.21: Teste de banda com links setados a 10GB, UDP, delay 1ms em topologia FatTree.

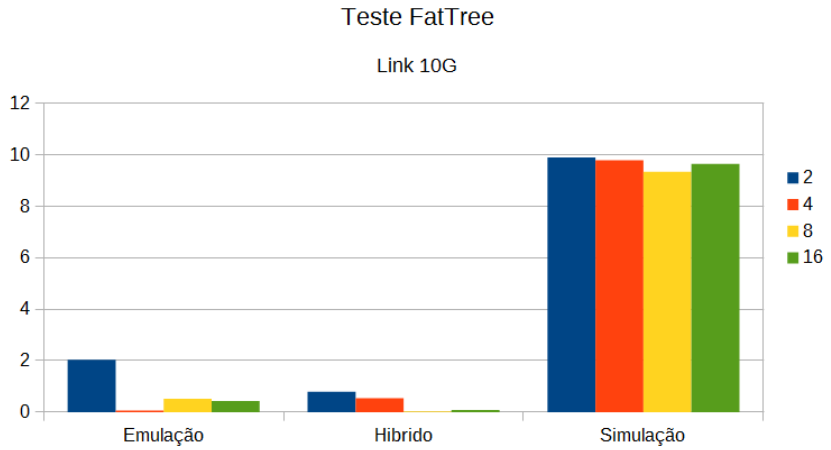

Figura 3.22: Teste de banda com links setados a 10GB, TCP, delay 1ms em topologia FatTree.

A queda de banda entre os tipos de protocolo é esperada, mas existe a manutenção de uma relação de comparação sistêmica com os testes em UDP. Apesar da necessidade da confirmação, a aplicação OpenFlow não influi no resultado, tornando-se parte estática dos fluxos a serem processados pelos *switches*. A banda disponível diminui ainda mais e algumas pesquisas (AL-FARES, 2010) utilizam diretamente o TCP como protocolo da camada de transporte. É claro que a simulação apresenta vantagens interessantes, entretanto há problemas antigos de modelagem. Outro gráfico levantado foi a quantidade de tempo para cada rodada de execução dos testes:

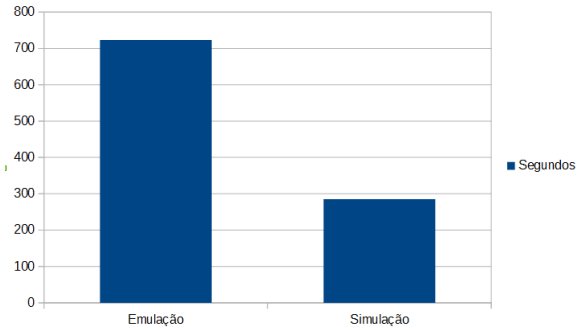

Figura 3.23: Tempo necessário para a realização de cada rodada.

Como a plataforma híbrida não possui uma forma automatizada de testes, o tempo não entrou neste gráfico.

Capítulo 4

CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho se propôs a demonstrar, em diversos aspectos, as trocas de desempenho entre propostas que utilizem OpenFlow em sua pesquisa e quais plataformas podem entregar uma alta fidelidade de resultados, com baixo custo. O levantamento dos dados do capítulo anterior deixa claro que, dependendo do tipo de ambiente almejado, existem limitações evidentes, tanto para emulação quanto simulação. Os ganhos e perdas são contrapontos difíceis, sem um meio termo que consiga agregar todas as vantagens de cada uma das tecnologias.

A comparação da queda de banda por meio da quantidade de hosts adicionados, utilizando toda a banda disponível, bem como a demora na execução de uma simulação para entrega do ambiente sem alterações, aponta que a proposta híbrida pode agregar de forma quase que exponencial as pesquisas realizadas no universo da experimentação. É um ponto bem conhecido a facilidade do uso de aplicações pessoais em ambientes emulados pela partilha do sistema base e *containers*, além das limitações de hardware que podem influenciar ativamente nos resultados. Na simulação, apesar de não ter o problema de limitação do hardware, de forma a influenciar nos resultados, pesam as questões de implementação, que podem ser resolvidas pela inserção de emulação. Os resultados demonstram que, para utilização de uma aplicação desenvolvida em OpenFlow, se o tráfego de dados não for um requisito primordial e a quantidade de hosts não for muito alta, os resultados podem ser similares entre as plataformas. Porém, ainda há de se ressaltar a dificuldade extrema em desenvolver uma aplicação OpenFlow em simulação, utilizando NS-3.

Os gráficos 3.6, 3.7 e 3.8 mostram que existe uma limitação de hardware clara para o uso de emulação, com links que almejam atingir grandes transmissões de dados. A figura 3.18, representa uma realidade dentro de *datacenters*, em que as conexões de dados de 10Gigabits não são tão estranhas. As Redes Definidas por Software têm sido bem aceitas em diversos tipos de implantações deste tipo de layout mas, desenvolver aplicações sem um devido teste

pode comprometer a entrega de uma pesquisa ou produto. Nesse sentido, a simulação mostra-se interessante, visto que o recurso de hardware requerido é quase nulo, perto do consumo e quantidade de tempo gastos pela emulação ao realizar a mesma tarefa. Vale ressaltar que existem as diferenças de consumo de memória e processamento que alteram a entrega final dos resultados, detalhe esse que pode ser interessante se for implementado em simulações, com o intuito de limitar e aproximar os recursos reais dos simulados.

O método híbrido entrega uma promissora relação que ainda necessita ser melhor explorada por recursos como o DCE, que inclui de maneira facilitada uma aplicação dentro do ambiente simulado, sem a necessidade de modelar o mesmo. Ideias nesse sentido de união entre as formas de utilização de *testbeds*, simulação e emulação podem ser um futuro promissor, vistas as qualidades apresentadas por cada modelo. Os bons resultados com pouca interferência do ambiente na aplicação OpenFlow necessitam de um desenvolvimento árduo, enquanto as aplicações prontas sofrem com as limitações de hardware.

Para uma melhor utilização de todos os recursos disponíveis, além do cuidado de escolha, é necessário prever o tamanho do layout desejado. As plataformas, cada qual com seus problemas, devem, em um futuro próximo, migrar para um uso conjunto que solucione os problemas paralelos, adicionando novos recursos e formas de utilização ao ambiente híbrido. Um switch emulado, por exemplo, tem disponível todo processamento, a menos que seja especificado que o mesmo não seja utilizado, entretanto, a memória está sempre disponível (HUANG; YOCUM; SNOEREN, 2013). Ainda é necessário que exista uma aproximação do uso de recursos para entrega de dados ainda mais fidedignos.

4.0.1 Trabalhos futuros

Discutido entre os resultados, uma forma híbrida de utilização pode auxiliar a simulação a criar um ambiente perfeito para pesquisa. Ainda em simulações, não é possível modificar a quantidade de processamento ou memória a ser utilizada, os aspectos de hardware hoje são pouco explorados dentro do ambiente simulado, de forma que a simulação sempre terá o melhor caso possível para a experimentação. Gargalos de processamento, disco, fila de pacotes ou limitações nas transmissões, ainda não foram adicionados de forma que seja possível determinar que um host utilize apenas metade de seu processamento. As *bridges* ou switches utilizados tanto na emulação quanto na simulação ainda não entregam dados que possam ser comparados com um switch comum utilizado em uma *testbed*, por exemplo.

Adicionar formas de controle de consumo de hardware nos objetos simulados além de adicionar formas mais simples de inserção de aplicativos em ambiente simulado são contribuições

positivas a serem estudadas e trabalhadas nos trabalhos futuros.

REFERÊNCIAS

AFANASYEV, A. et al. ndnsim: Ndn simulator for ns-3. 2012.

AL-FARES, M.; LOUKISSAS, A.; VAHDAT, A. A scalable, commodity data center network architecture. In: ACM. *ACM SIGCOMM Computer Communication Review*. [S.l.], 2008. v. 38, p. 63–74.

AL-FARES, M. et al. Hedera: Dynamic flow scheduling for data center networks. In: *NSDI*. [S.l.: s.n.], 2010. v. 10, p. 19–19.

BAKE: An integration tool for NS-3. http://code.nsnam.org/daniel/bake/. Accessed: 2014-10-30.

BARI, M. F. et al. Policycop: An autonomic qos policy enforcement framework for software defined networks. In: IEEE. *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. [S.l.], 2013. p. 1–7.

CARNEIRO, G.; FORTUNA, P.; RICARDO, M. Flowmonitor: a network monitoring framework for the network simulator 3 (ns-3). In: ICST (INSTITUTE FOR COMPUTER SCIENCES, SOCIAL-INFORMATICS AND TELECOMMUNICATIONS ENGINEERING). *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*. [S.l.], 2009. p. 1.

CHONG, E. K. Discrete event systems: Modeling and performance analysis. *Discrete Event Dynamic Systems*, Springer, v. 4, n. 1, p. 113–116, 1994.

CONSORTIUM, O. S. et al. *OpenFlow Switch Specification Version 1.0. 0*. [S.l.]: December, 2009.

DENAZIS, S. et al. Software-defined networking (sdn): Layers and architecture terminology. 2015.

DOXYGEN. http://www.stack.nl/ dimitri/doxygen/. Accessed: 2014-10-30.

FIBRE. http://www.fibre-ict.eu/. Accessed: 2014-10-30.

FP7, O. *Ofelia Control Framework*. Disponível em: <http://fp7-ofelia.github.io/ocf/>.

GUPTA, M.; SOMMERS, J.; BARFORD, P. Fast, accurate simulation for sdn prototyping. In: ACM. *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. [S.l.], 2013. p. 31–36.

HP-VAN Controller. https://www.hpe.com/us/en/networking/sdn.html. Accessed: 2016-10-30.

HU, F.; HAO, Q.; BAO, K. A survey on software defined networking (sdn) and openflow: From concept to implementation. IEEE, 2014.

HUANG, D. Y.; YOCUM, K.; SNOEREN, A. C. High-fidelity switch models for software-defined network emulation. In: ACM. *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. [S.l.], 2013. p. 43–48.

JAIN, R.; PAUL, S. Network virtualization and software defined networking for cloud computing: a survey. *IEEE Communications Magazine*, IEEE, v. 51, n. 11, p. 24–31, 2013.

JAIN, S. et al. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 43, n. 4, p. 3–14, ago. 2013. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2534169.2486019>.

JANSEN, S.; MCGREGOR, A. Simulation with real world network stacks. In: IEEE. *Simulation Conference, 2005 Proceedings of the Winter*. [S.l.], 2005. p. 10–pp.

JONDRAL, F. K. Software-defined radio: Basics and evolution to cognitive radio. *EURASIP J. Wirel. Commun. Netw.*, Hindawi Publishing Corp., New York, NY, United States, v. 2005, n. 3, p. 275–283, ago. 2005. ISSN 1687-1472. Disponível em: <http://dx.doi.org/10.1155/WCN.2005.275>.

JOY, A. M. Performance comparison between linux containers and virtual machines. In: IEEE. *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*. [S.l.], 2015. p. 342–346.

KLEIN, D.; JARSCHEL, M. An openflow extension for the omnet++ inet framework. In: ICST (INSTITUTE FOR COMPUTER SCIENCES, SOCIAL-INFORMATICS AND TELECOMMUNICATIONS ENGINEERING). *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. [S.l.], 2013. p. 322–329.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015.

KULDEEP K. SHARMA, M. S. Mininet as a container based emulador for software defined networks. 2014.

LACAGE, M. *Experimentation tools for networking research*. Tese (Doutorado) — Ph. D. dissertation, Ecole doctorale Stic, Université de Nice Sophia Antipolis, 2010.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: ACM. *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. [S.l.], 2010. p. 19.

LUO, T.; TAN, H.-P.; QUEK, T. Q. Sensor openflow: Enabling software-defined wireless sensor networks. *IEEE Communications Letters*, IEEE, v. 16, n. 11, p. 1896–1899, 2012.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

MININET An Instant Virtual Network on your Laptop (or other PC). http://mininet.org/. Accessed: 2016-10-30.

MININET API Reference Guide. http://mininet.org/api/hierarchy.html. Accessed: 2016-12-20.

NASCIMENTO, M. R. et al. Routeflow: Roteamento commodity sobre redes programáveis. *XXIX Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuıdos-SBRC*, 2011.

NIKOLAEV, S. et al. Performance of distributed ns-3 network simulator. In: ICST (INSTITUTE FOR COMPUTER SCIENCES, SOCIAL-INFORMATICS AND TELECOMMUNICATIONS ENGINEERING). *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. [S.l.], 2013. p. 17–23.

NOXREPO.ORG. *POX/NOX*. Disponível em: <http://www.noxrepo.org/pox/about-pox/>.

NS 3. http://www.nsnam.org/. Accessed: 2014-10-30.

NS-3 Network Simulator Tutorial Release ns-3.26. https://www.nsnam.org/docs/release/3.26/tutorial/ns-3-tutorial.pdf. Accessed: 2016-10-30.

NS-3 Openflow Integration. http://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html. Accessed: 2014-10-30.

NUNES, B. et al. A survey of software-defined networking: Past, present, and future of programmable networks. IEEE, 2014.

OMNET. *OMNeT ++ Website*. Disponível em: <http://omnetpp.org/>.

ONF. *Open Network Foundation Website*. Disponível em: <https://www.opennetworking.org/>.

OPEN vSwitch. http://openvswitch.org/. Accessed: 2014-10-30.

PFAFF, B. et al. Openflow switch specification, version 1.3. 0. *Open Networking Foundation*, 2012.

POX networking software platform written in Python. https://github.com/noxrepo/pox. Accessed: 2016-10-30.

ROTTENBERG, C. E. L. F. C. *OpenFlow 1.3 Soft Switch*. Disponível em: <http://sbrc2014.ufsc.br/anais/files/salao/SF-ST3-1.pdf>.

RYU. http://osrg.github.io/ryu/. Accessed: 2015-1-10.

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

SHIN, S. et al. Fresco: Modular composable security services for software-defined networks. In: *NDSS*. [S.l.: s.n.], 2013.

SHIN, S. et al. Avant-guard: Scalable and vigilant switch flow management in software-defined networks. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*. New York, NY, USA: ACM, 2013. (CCS '13), p. 413–424. ISBN 978-1-4503-2477-9. Disponível em: <http://doi.acm.org/10.1145/2508859.2516684>.

SOMMERS, J. et al. Efficient network-wide flow record generation. In: IEEE. *INFOCOM, 2011 Proceedings IEEE*. [S.l.], 2011. p. 2363–2371.

TAZAKI, H. et al. Direct code execution: Realistic protocol simulation with running code. 2013.

TAZAKI, H.; URBANI, F.; TURLETTI, T. Dce cradle: simulate network protocols with real stacks for better realism. In: ICST (INSTITUTE FOR COMPUTER SCIENCES, SOCIAL-INFORMATICS AND TELECOMMUNICATIONS ENGINEERING). *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. [S.l.], 2013. p. 153–158.

TECHNOLOGIES, E. *EstiNet Website*. Disponível em: <http://www.estinet.com/>.

VBOXMANAGE. https://www.virtualbox.org/manual/ch08.html. Accessed: 2016-10-30.

WANG, S.-Y. Comparison of sdn openflow network simulator and emulators: Estinet vs. mininet. In: IEEE. *Computers and Communication (ISCC), 2014 IEEE Symposium on*. [S.l.], 2014. p. 1–6.

Appendices

Apêndice A

Apêndice A

```
#include <iostream>
#include <fstream>

#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/applications-module.h"
#include "ns3/bridge-module.h"
#include "ns3/csma-module.h"
#include "ns3/internet-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE ("CSMA Bridge Example");

int main (int argc, char *argv[]){

LogComponentEnable ("UdpTraceClient", LOG_LEVEL_INFO);
LogComponentEnable ("UdpServer", LOG_LEVEL_INFO);


Address serverAddress;


CommandLine cmd;
```

```
cmd.Parse (argc, argv);

NodeContainer terminalsS1;
terminalsS1.Create (2);

NodeContainer csma_switch1;
csma_switch1.Create(1);

NodeContainer terminalsS2;
terminalsS2.Create (2);

NodeContainer csma_switch2;
csma_switch2.Create(1);

NodeContainer terminalsS3;
terminalsS3.Create (2);

NodeContainer csma_switch3;
csma_switch3.Create(1);

NodeContainer terminalsS4;
terminalsS4.Create (2);

NodeContainer csma_switch4;
csma_switch4.Create(1);



CsmaHelper csma;
csma.SetChannelAttribute ("DataRate", StringValue ("100Mbps"));
csma.SetChannelAttribute ("Delay", TimeValue (NanoSeconds (6500)));

NetDeviceContainer terminalDevices1;
NetDeviceContainer csma_switch_devices1;

NetDeviceContainer terminalDevices2;
```

```
NetDeviceContainer csma_switch_devices2;

NetDeviceContainer terminalDevices3;
NetDeviceContainer csma_switch_devices3;

NetDeviceContainer terminalDevices4;
NetDeviceContainer csma_switch_devices4;

for (int i = 0; i < 2; i++ ) {

NetDeviceContainer link1 = csma.Install (NodeContainer (terminalsS1.Get(i), csma_sw
terminalDevices1.Add (link1.Get(0));
csma_switch_devices1.Add(link1.Get(1));

NetDeviceContainer link2 = csma.Install (NodeContainer (terminalsS2.Get(i), csma_sw
terminalDevices2.Add (link2.Get(0));
csma_switch_devices2.Add(link2.Get(1));

NetDeviceContainer link3 = csma.Install (NodeContainer (terminalsS3.Get(i), csma_sw
terminalDevices3.Add (link3.Get(0));
csma_switch_devices3.Add(link3.Get(1));

NetDeviceContainer link4 = csma.Install (NodeContainer (terminalsS4.Get(i), csma_sw
terminalDevices4.Add (link4.Get(0));
csma_switch_devices4.Add(link4.Get(1));

}

NetDeviceContainer linkS12 = csma.Install (NodeContainer (csma_switch1.Get(0), csma
NetDeviceContainer linkS13 = csma.Install (NodeContainer (csma_switch1.Get(0), csma
NetDeviceContainer linkS14 = csma.Install (NodeContainer (csma_switch1.Get(0), csma

NetDeviceContainer linkS23 = csma.Install (NodeContainer (csma_switch2.Get(0), csma
NetDeviceContainer linkS24 = csma.Install (NodeContainer (csma_switch2.Get(0), csma
```

```
NetDeviceContainer linkS34 = csma.Install (NodeContainer (csma_switch3.Get(0), csma

csma_switch_devices1.Add(linkS12.Get(0));
csma_switch_devices2.Add(linkS12.Get(1));

csma_switch_devices1.Add(linkS13.Get(0));
csma_switch_devices3.Add(linkS13.Get(1));

csma_switch_devices1.Add(linkS14.Get(0));
csma_switch_devices4.Add(linkS14.Get(1));

csma_switch_devices2.Add(linkS23.Get(0));
csma_switch_devices3.Add(linkS23.Get(1));

csma_switch_devices3.Add(linkS34.Get(0));
csma_switch_devices4.Add(linkS34.Get(1));

Ptr<Node> switchNode1 = csma_switch1.Get(0);
Ptr<Node> switchNode2 = csma_switch2.Get(0);
Ptr<Node> switchNode3 = csma_switch3.Get(0);
Ptr<Node> switchNode4 = csma_switch4.Get(0);


Ptr<ns3::ofi::LearningController> controller = CreateObject<ns3::ofi::LearningContr
switchNode1.Install (switchNode1, switch1Devices, controller);
switchNode2.Install (switchNode2, switch2Devices, controller);
switchNode3.Install (switchNode3, switch1Devices, controller);
switchNode4.Install (switchNode4, switch2Devices, controller);

InternetStackHelper internet;
internet.Install (terminalsS1);
internet.Install (terminalsS2);

Ipv4AddressHelper ip;
ip.SetBase ("10.1.1.0", "255.255.255.0");
```

```
Ipv4InterfaceContainer i = ip.Assign (terminalDevices1);
serverAddress = Address (i.GetAddress(0));
ip.Assign(terminalDevices2);

uint16_t port = 4000;
UdpServerHelper server (port);
ApplicationContainer apps = server.Install (terminalsS1.Get (0));
apps.Start (Seconds (1.0));
apps.Stop (Seconds (10.0));

uint32_t MaxPacketSize = 1472;
UdpTraceClientHelper client (serverAddress, port,"");
client.SetAttribute ("MaxPacketSize", UintegerValue (MaxPacketSize));
apps = client.Install (terminalsS2.Get (0));
apps.Start (Seconds (2.0));
apps.Stop (Seconds (10.0));

Simulator::Run();
Simulator::Destroy();




return 0;
}
```

Apêndice B

Apêndice B

```
#!/usr/bin/python



from mininet.net import Mininet
from mininet.node import RemoteController, OVSSwitch
from mininet.cli import CLI as execute
from mininet.log import setLogLevel, info
import itertools


def emptyNet():

net = Mininet( topo=None, build=False, xterms=True )

info( '*** Adding controller\n' )
net.addController('c0', controller=RemoteController,ip="127.0.0.1",port=6633)


info( '*** Adding hosts\n' )
h1 = net.addHost( 'h1', ip='10.0.2.1' )
h2 = net.addHost( 'h2', ip='10.0.2.2' )
h3 = net.addHost( 'h3', ip='10.0.2.3' )
h4 = net.addHost( 'h4', ip='10.0.2.4' )
```

```
h5 = net.addHost( 'h5', ip='10.0.2.5' )
h6 = net.addHost( 'h6', ip='10.0.2.6' )
h7 = net.addHost( 'h7', ip='10.0.2.7' )
h8 = net.addHost( 'h8', ip='10.0.2.8' )

h9 = net.addHost( 'h9', ip='10.0.2.9' )
h10 = net.addHost( 'h10', ip='10.0.2.10' )
h11 = net.addHost( 'h11', ip='10.0.2.11' )
h12 = net.addHost( 'h12', ip='10.0.2.12' )

h13 = net.addHost( 'h13', ip='10.0.2.13' )
h14 = net.addHost( 'h14', ip='10.0.2.14' )
h15 = net.addHost( 'h15', ip='10.0.2.15' )
h16 = net.addHost( 'h16', ip='10.0.2.16' )

h17 = net.addHost( 'h17', ip='10.0.2.17' )
h18 = net.addHost( 'h18', ip='10.0.2.18' )
h19 = net.addHost( 'h19', ip='10.0.2.19' )
h20 = net.addHost( 'h20', ip='10.0.2.20' )

h21 = net.addHost( 'h21', ip='10.0.2.21' )
h22 = net.addHost( 'h22', ip='10.0.2.22' )
h23 = net.addHost( 'h23', ip='10.0.2.23' )
h24 = net.addHost( 'h24', ip='10.0.2.24' )

h25 = net.addHost( 'h25', ip='10.0.2.25' )
h26 = net.addHost( 'h26', ip='10.0.2.26' )
h27 = net.addHost( 'h27', ip='10.0.2.27' )
h28 = net.addHost( 'h28', ip='10.0.2.28' )

h29 = net.addHost( 'h29', ip='10.0.2.29' )
h30 = net.addHost( 'h30', ip='10.0.2.30' )
h31 = net.addHost( 'h31', ip='10.0.2.31' )
h32 = net.addHost( 'h32', ip='10.0.2.32' )
```

```
info( '*** Adding switch\n' )
s1 = net.addSwitch( 's1', cls=OVSSwitch )
s2 = net.addSwitch( 's2', cls=OVSSwitch )
s3 = net.addSwitch( 's3', cls=OVSSwitch )
s4 = net.addSwitch( 's4', cls=OVSSwitch )

info( '*** Creating links\n' )

net.addLink( h1, s1 )
net.addLink( h2, s3 )
net.addLink( h3, s2 )
net.addLink( h4, s4 )

net.addLink( h5, s1 )
net.addLink( h6, s3 )
net.addLink( h7, s2 )
net.addLink( h8, s4 )

net.addLink( h9, s1 )
net.addLink( h10, s3 )
net.addLink( h11, s2 )
net.addLink( h12, s4 )

net.addLink( h13, s1 )
net.addLink( h14, s3 )
net.addLink( h15, s2 )
net.addLink( h16, s4 )

net.addLink( h17, s1 )
net.addLink( h18, s3 )
net.addLink( h19, s2 )
net.addLink( h20, s4 )
```

```
net.addLink( h21, s1 )
net.addLink( h22, s3 )
net.addLink( h23, s2 )
net.addLink( h24, s4 )

net.addLink( h25, s1 )
net.addLink( h26, s3 )
net.addLink( h27, s2 )
net.addLink( h28, s4 )

net.addLink( h29, s1 )
net.addLink( h30, s3 )
net.addLink( h31, s2 )
net.addLink( h32, s4 )




switchList = (s1, s2, s3, s4)
for index in range (0, len(switchList)):
for index2 in range (index+1, len(switchList)):
net.addLink(switchList[index], switchList[index2])

info( '*** Starting network\n')
net.start()



s1.cmd('ifconfig s1 10.0.1.1')
s2.cmd('ifconfig s2 10.0.1.2')
s3.cmd('ifconfig s3 10.0.1.3')
s4.cmd('ifconfig s4 10.0.1.4')

info('*** Switches\n')
s1.cmd('ovs-vsctl set bridge s1')
```

```
s2.cmd('ovs-vsctl set bridge s2 ')
s3.cmd('ovs-vsctl set bridge s3 ')
s4.cmd('ovs-vsctl set bridge s4 ')

info( '*** Running CLI\n' )
test = execute(net )
test.pingall()
test(CLI)


info( '*** Stopping network' )
net.stop()


if __name__ == '__main__':
setLogLevel( 'info' )
emptyNet()
```

GLOSSÁRIO

CAPEX – *Capital Expenditure*

CBE – *Container Based Emulation*

CBR – *Constant Bit Rate*

DCE – *Direct Code Execution*

DES – *Discrete Event System*

IDS – *Intrusion Detection System*

NFV – *Network Function Virtualization*

NSC – *Network Simulation Cradle*

OCF – *Ofelia Control Framework*

OPEX – *Operational Expenditure*

OVS – *Open VSwitch*

SDN – *Software Definied Network*

VETH – *Virtual Ethernet*