

# 山东大学

## 毕业论文(设计)

论文（设计）题目：

SDN 环境下的虚拟网络搭建

姓 名 张洪利

学 号 201400301181

学 院 山东大学软件学院

专 业 软件工程

年 级 2014 级

指导教师 朱方金

2018 年 5 月 5 日

年级:

学号		姓名		设计（论文）成绩	
设计（论文）题目					
指导教师评语					
	评定成绩：		签名：		年 月 日
评阅人评语					
	评定成绩：		签名：		年 月 日
答辩小组评语					
	答辩成绩：		组长签名：		年 月 日

注：设计（论文）成绩=指导教师评定成绩（30%）+评阅人评定成绩（30%）+答辩成绩（40%）

## 目录

摘    要.....	4
ABSTRACT.....	5
第 1 章 绪论.....	7
1.1 SDN 技术的本质及背景.....	7
1.2 国内外研究现状 .....	7
1.3 解决的主要问题 .....	8
1.4 本文的主要工作 .....	9
1.5 论文的组织结构 .....	9
第 2 章 开发工具选择及环境搭建.....	10
2.1 开发工具的选择 .....	10
2.1.1 实验功能简析.....	10
2.1.2 确定开发工具.....	10
2.2 开发环境搭建 .....	11
2.2.1 Mininet2.3.0 安装.....	11
2.2.2 Open vSwitch-2.3.0 安装.....	12
2.2.3 sFlow-rt 流量监控环境搭建 .....	15
第 3 章 实验目标分析和功能分析.....	19
3.1 实验目标分析 .....	19
3.1.1 虚拟网络拓扑规模.....	19
3.1.2 虚拟网络拓扑结构.....	20
3.2 实验功能分析 .....	20
3.2.1 虚拟网络拓扑搭建方式.....	20
3.2.2 Mininet 视图下对网络结构的二次重构 .....	22
第 4 章 实验原理剖析及实现方法分析.....	23
4.1 实验原理剖析 .....	23
4.1.1 SDN 与网络虚拟化 .....	23
4.1.2 Mininet 功能逻辑 .....	24
4.2 Mininet 源码解读.....	26

4.2.1 Mininet 源码结构分析 .....	26
4.2.2 核心代码 net.py 解读.....	28
4.3 核心功能实现 .....	31
4.3.1 网络拓扑初始化.....	31
4.3.2 批量删除网络链路.....	33
4.3.3 批量删除网络节点.....	35
4.3.4 网络性能测试及流量发送.....	36
4.3.5 Mininet 命令扩展.....	37
第 5 章 实验方法与结果分析.....	40
5.1 虚拟网络拓扑创建 .....	40
5.2 虚拟网络拓扑重构 .....	42
5.2.1 网络链路删除测试.....	42
5.2.2 网络节点删除测试.....	44
第 6 章 结论.....	46
致谢.....	47
参考文献.....	48
附录 1 英文原文 .....	49
附录 2 译文 .....	52

## SDN 环境下虚拟网络的搭建

### 摘 要

虚拟化技术是一种使用软件的方法重新定义和划分 IT 资源，虚拟化和实现 IT 资源动态分配和灵活调度，以此来达到提高 IT 资源的利用率的目的，是 IT 资源真正成为社会的基础设施。而且通过对虚拟化技术的概念和技术的扩展可以使其服务于各行各业灵活多变的需求。

网络虚拟化、软件定义网络、智慧数据等“新鲜”名词随着信息时代的快速发展慢慢进入学者甚至大众的视野，它们同属于虚拟化技术的分支，虚拟网

络就是网络虚拟化中的产物。

软件定义网络 (Software Defined Network, SDN ), 是一种新型网络创新架构, 是网络虚拟化的一种实现方式, 其核心技术 OpenFlow 通过将网络设备控制面与数据面分离开来, 从而实现了网络流量的灵活控制, 使网络作为管道变得更加智能。

在一个计算机网络中如果其中的链路或者部分链路是虚拟网络链接, 那么这个计算机网络就可以成为虚拟网络。虚拟网络内部节点之间的链接是通过网络虚拟化技术实现的, 任意两台计算设备之间不存在真实的物理连接。例如基于协议的 VPN (Virtual Private Network, 虚拟专用网络)、VLAN (Virtual Local Area Network, 虚拟局域网) 和 VPLS (Virtual Private Lan Service, 虚拟专用局域网业务), 以及基于虚拟设备的 (如 hypervisor) 的虚拟网络。

虚拟网络的原理是对传统的物理网络 (以太网) 使用隧道技术 (例如 VXLAN、NVGRE 提供的 3 层网络隧道) 对其进行隔离、划分, 并允许虚拟机在数据中心内的网络间或者在数据中心之间迁移。

本文旨在通过 SDN 对虚拟网络的支持, 探讨利用软件技术单间虚拟网网络的方法。通过现有的 SDN 工具完成多要求的虚拟网络拓扑结构的搭建, 并基于此搭建好的网络拓扑进行一系列测试, 统计、分析实验现象和实现数据, 最后得出结论。

**关键字:** 虚拟化技术、网络虚拟化、计算机网络、SDN

## ABSTRACT

Virtualization technology is a method of using software to redefine 、divide and virtualize IT resources, achieve the goal that allocation dynamically and flexibly schedule IT resources, in order to achieve the purpose of improving the utilization of IT resources and make sure that IT resources really become the basis of society facility. What' s more, through the extension of virtualization technology concepts and technologies, it can serve the flexible needs of all walks of life.

The new terms such as network virtualization, software-defined networks, and smart data slowly enter the eyes of scholars and even the general public with the rapid development of the information age. They belong to the branch of virtualization technology. Virtual networks are the products of network virtualization.

SDN (Software Defined Network), it is a new type of network innovation architecture. It is an implementation method of network virtualization. Its core technology, OpenFlow, separates the control plane of the network equipment from the data plane, thus enabling flexible control of network traffic and making the network change as a pipeline smarter.

In a computer network, if the link or part of the link is a virtual network link, then the computer network can become a virtual network. The virtual network link does not include the actual physical connection between the two computing devices, but through the network. Virtualization to achieve. Such as VPN (Virtual Private Network) which based on protocol, VLAN (Virtual Local Area Network)、VPLS (Virtual Private Lan Service) and virtual network which based on internet device, such as hypervisor.

The principle of the virtual network is to use a tunneling technology to create a virtual network over a traditional physical network (Ethernet), to use a protocol such as VXLAN or NVGRE to provide a layer 3 network tunnel, and to allow virtual machines to be located in networks or data within a data center. Migration between centers.

This article aims to explore the use of software technology for single-site virtual network networks through SDN support for virtual networks. Through the existing SDN tools to complete the requirements of the virtual network topology structure, and based on this set up a good network topology for a series of tests, statistics, analysis of experimental phenomena and data, and finally reached a conclusion.

**Key Words:** Virtualization Technology, Network Virtualization, Computer Networks, SDN

## 第 1 章 绪论

### 1.1 SDN 技术的本质及背景

传统的 IP 网络经过 30 多年的发展已经从最初只能满足简单的 Internet 服务的网络，演进成了能够提供多媒体业务（文本、语音、图像、视频等）的融合网络。而网络的应用领域也在慢慢的想生活的各个方面渗透，对人们的生产和生活方式也产生了潜移默化的影响。但是随着互联网业务的发展，网络也面临着一系列新的问题：网络设备日趋复杂、管理运维复杂、网络创新难、业务部署周期长（通常需要 3-5 年）等。这些问题严重制约了网络技术和网络业务的扩展，成为新业务发展的瓶颈。

为了解决上述的问题，学术界和业界一直在探索能够用来提升网络灵活性，打破网络的封闭架构，增强网络可编程能力的技术方案。经过多年的研究和探索，SDN 应运而生。

SDN 字面意思是软件定义网络（Software Defined Network），是一种试图摆脱硬件对网络架构的限制的技术，通过 SDN 可以像升级、安装软件一样对网络进行修改，使得更多的应用程序能够方便、快速部署到网络上。

SDN 的本质是网络软件化，能提升网络可编程能力，比原来网络架构更好、更快、更简单的实现各种定制功能特性。SDN 是一次网络架构的重构，而不是一种新特性或新功能。

### 1.2 国内外研究现状

2011 年，SDN 的研究者成立了 ONF<sup>[1]</sup>（开放式网络基金会，Open Networking Foundation），专门负责相关标准的制定和推广，使 SDN<sup>[2]</sup>成为了全球开放网络架构和网络虚拟化领域的研究热点。自此，在学术界和产业界掀起了一股 SDN 的研究浪潮。

在国外，美国、欧洲、日本先后开展对 SDN 的研究和部署，许多基于 SDN 产品的创业公司开始不断涌现，例如被 VMware<sup>[3]</sup> 收购的 Nicira、BigSwitch 等，一些老牌的网络硬件巨头如 NEC、IBM、Cisco 等也先后发布了支持 SDN 的硬件产品。

而在国内，由清华大学牵头，中科院计算所、北邮、东南大学、北京大学等单位参与开展了类 SDN 思想的“未来网络体系结构和创新环境<sup>[4]</sup>”863 项目研究。华为、中兴等设备厂商，中国移动、中国电信等运营商都相继开发自己基于 SDN 技术的敏捷控制器；百度、腾讯、阿里巴巴等网络厂商也都相继加入了 ONF，积极参与 SDN 技术的研究和产品解决方案的开发。

可见，国外的 SDN 技术发展迅速，已经较早的进入了商用部署阶段，而国内研究结构、高校、和厂商虽然起步较晚，但是也紧跟国际步伐，慢慢的从实验阶段过渡到了试点应用阶段。发展形势一片大好。

### 1.3 解决的主要问题

传统的网络架构虽然生存能力很强，但是随着技术的发展和网络业务的不断扩展，网络架构本身变得越来越复杂。使得网络应用的部署周期大大延长，部署成本大大提高，恰恰这是所有人都不能忍受的。

形象一点的讲，现在网络设备复杂，要搭建一个网络架构需要精通各种不同网络设备的配置方法，然后按部就班的对各种设备进行配置，如果网络结构的体量过大，其工作量可想而知，而且出现错误的可能性太大，出错后不易反向和修正。

通过 SDN 技术，颠覆了传统网络部署技术，使得网络的控制面板和转发面分离，网络工程师不再需要跟大量令人头疼的命令打交道，只需要在可视化界面上对设备进行批量配置之后，有 SDN 控制器<sup>[5]</sup>（如 ODL<sup>[6]</sup>、敏捷控制器等）自动下发配置命令，而且可以自动检测网络部署过程中出现的错误。减少了应用程序部署的人力和时间成本。更重要的是提高了网络的灵活性以及开放的网络可编程能力，在结合其他虚拟化技术大大简化了网络功能定制化的难度。

SDN 是对传统网络架构的一种创新。



## 1.4 本文的主要工作

本文旨在研究如何利用 SDN 技术搭建虚拟网络<sup>[7]</sup>，利用现有的 SDN 研究工具来方便的进行特定结构虚拟网络的搭建。利用搭建好的虚拟网络拓扑结构来雁阵虚拟网络的结构是否可靠。只要方法是对于建立起来的虚拟网络拓扑进行一些网络节点或者网络链路的删除操作，剩余的部分是一个完整的与原先的结构有所不同的新的虚拟网络拓扑，再在新的拓扑结构的基础上进行网络结构的完整性的正确性的验证。实验的目的是证明虚拟网络的修改不会破坏原有的网络结构，即产出节点和链路候只对与被删除掉节点和链路有直接关联的节点和虚拟子网有影响，其余子网的连通性并不受影响。

## 1.5 论文的组织结构

论文正文共分为 6 个章节。

第 1 章为绪论。主要介绍了 SDN 的产生的背景和发展历史、国内外的研究现状以及 SDN 需要解决的问题。同时还对本次毕业设计的主要工作做了简单的介绍。

第 2 章为开发工具的选择及环境搭建。本章通过对毕业设计的功能性需求和实验目的的简单分析确定了在实验过程中需要用到的模拟工具和需要搭建的开发、模拟环境。以及主要工具的安装、构建方法。

第 3 章实验目标分析和功能分析。明确了实验需要搭建的网络拓扑的结构和规模。根据模拟工具的开放 API 文档确定了虚拟网络拓扑的搭建、重构的操作方式以及代码实现的可行性。

第 4 章为实验原理剖析和实验方法分析。详细介绍了本次实验主要用到的工具 Mininet 的功能逻辑、源码结构以及核心模块的源码分析。为后面的具体开发做准备，本章的最后一节详细介绍了本次实验的核心开发工作以及实现方式。

第 5 章为虚拟软件定义网络测试。即笔者开发的函数的使用方法，详细介绍了实验步骤并利用自己的代码和实验结果验证本文的论点。

第 6 章为结论。主要通过对实验现象和实验结果的阐述来论证本文观点

## 第 2 章 开发工具选择及环境搭建

在进行毕业设计的正式开发和实验之前，我们需要先明确实验的目标和方向，并根据具体功能选择合适的开发工具。一个合适的工具的使用会使得我们的工作事半功倍。本章在简单明确了实验内容和工作的情况之后确定了我们需要的工具，并详细介绍了各种工具的安装方法。

### 2.1 开发工具的选择

#### 2.1.1 实验功能简析

本课题的基本要求是利用现有的 SDN 研究工具来方便的进行特定结构的虚拟网络的搭建。利用搭建好的虚拟网络拓扑结构来验证虚拟网络的结构是否可靠。验证方法主要是对建立起来的虚拟网络拓扑进行一些网络节点或者网络链路的删除操作，剩余的部分是一个与原先的结构不同的新的虚拟网络拓扑，再在新的拓扑结构的基础上进行网络结构的完整性、正确性的验证。

首先，要完成课题要求的虚拟网络，需要搭建一个存在 15-20 虚拟交换机节点的网络拓扑，因此需要有工具支持交换机的基本功能。

虚拟网络拓扑应该与实际网络结构高度相似，内部有相应的工作节点和网络通路，支持节点之间的连接测试（如 ping、package 转发）等，而且支持在虚拟网络运行场景下对网络结构本身进行更改。

环境搭建完成后需要检测网络内部的流量，证明在实验测试时虚拟网络拓扑内部有流量的传递，以此来支持实验结论的说服力。

#### 2.1.2 确定开发工具

使用 Openvswitch<sup>[8]</sup>（以下简称 OVS）作为虚拟交换机工具。OVS 支持标准管理接口和协议（如 NetFlow、sFlow、CLI 等），而且它通过软件的方式构成交换部件，比传统的物理交换机配置起来更加灵活，更方便新手上手，非常适合仿真环境。目前 OVC 已经被多种虚拟化平台和交换芯片所支持。

Mininet<sup>[9]</sup> 是一个基于 Linux Container 架构开发的虚拟化平台，通过

Mininet 可以轻松地在自己的笔记本上测试一个软件定义网络。一行命令就可以创建一个简单的三节点的软件定义网络，而且能够在网络拓扑运行时进行多种操作。如果 Mininet 提供的网络拓扑结构不足以满足你的实验要求，可以通过它提供的 Python API<sup>[10]</sup>以编写脚本的方式创建自定义的网络拓扑结构。非常符合课题要求的目标。而且 Mininet 支持 OpenFlow、sFlow。

sFlow 是基于标准的最新网络导出协议，不同于数据包采样技术，sFlow<sup>[11]</sup>是一种导出格式，以设备端口为基本单元的数据流随机采样技术。sFlow-rt 是一种 sFlow 流量监控工具，由 sFlow Agent 和 sFlow Collector 两部分组成。一般内嵌于网络转发设备，而且 Mininet 和 OVS 均提供了对 sFlow 的支持。

## 2.2 开发环境搭建

### 2.2.1 Mininet2.3.0 安装

Mininet 采用轻量级的虚拟化技术使一个单一系统看起来像一个真真实实运行的网络系统，是一个轻量级的软件定义网络和测试平台。Mininet 是用 Python<sup>[12]</sup>语言编写的软件定义网络模拟和测试工具<sup>[1]</sup>，笔者将在自己创建的 ubuntu 虚拟机上搭建实验环境。Ubuntu 版本为 Ubuntu 14.04.5 LTS，Linux 内核版本为 3.13.0-32-generic。Mininet 版本为 2.3.0。为了防止出现一些未知问题影响实验进程，建议在干净的 ubuntu 虚拟机上安装 Mininet 工具，如果之前安装过其他版本的 Mininet 工具，需要先将之前安装的 Mininet 工具卸载干净，然后在安装需要的 Mininet 版本。因为笔者是在自己刚刚安装好的 ubuntu 虚拟机上安装 Mininet2.3.0，所以不需要考虑这个问题。

首先，打开虚拟机 terminal 终端需要给当前用户获取 root 权限：

```
sudo -s;
```

输入当前用户密码，切换权限。之后执行以下命令：

```
apt-get update;
apt-get upgrade;
```

在安装 Mininet 之前，需要先更新包管理器，确保 Mininet 的版本是自己希望的版本。更新完包管理器之后需要将 Mininet 的源码下载到本地。由于

Mininet 是一个开源项目，其源码可在 github 上下载，仅需执行以下命令：

```
git clone git://github.com/mininet/mininet
```

执行命令之前需要确保虚拟机已经安装 git。源码下载成功后，可以通过“cat INSTALL”查看当前代码版本，或者通过“git tag”列出所有可用版本：

之后，通过 cd 命令进入“/mininet/util”，执行目录下的 install.sh 以安装 mininet 工具。执行“./install.sh -a”安装 Mininet 以及所有的依赖关系。这个过程需要花费一点时间，安装时提示信息比较多，此处笔者不再贴图，可自行根据提示信息判断是否完成安装。

安装完成后，可以通过执行“sudo mn”命令来建立一个 Mininet 自带的最简单的网络拓扑：两台主机 host (h1, h2)，一台交换机 switch (s1)。

```
root@zhl-virtual-machine:~/mininet# sudo mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet>
```

如果出现了以上 mininet>视图，则 Mininet2.3.0 安装完成。

### 2.2.2 Open vSwitch-2.3.0 安装

SDN 交换机是搭建 SDN 环境不可缺少功能组件之一，它与传统交换机最大的区别就是它将普通交换机的数据平面和控制平面分离，由更上一级的控制器下发控制指令，SDN 交换机只负责数据的转发。

Open vSwitch 交换机是一个高质量、多层次的虚拟交换机，支持多种标准协议，如 openFlow、sFlow 等。但是 OVS 的依赖关系比较多，功能选择也比较

灵活，因此安装过程比较麻烦，遇到的问题也会比较多。

和安装 Mininet 时一样，在正式安装 OVS 之前，需要在虚拟机上安装一些必需的依赖和库文件，获取 root 权限后，执行以下命令会自动安装库文件以及所需依赖：

```
# apt-get update
# apt-get install -y build-essential
```

准备工作完成之后，就可以开始正式部署安装 Open vSwitch-2.3.0 了。第一步需要获取 OVS 2.3.0 安装包：

```
# wget http://openvswitch.org/releases/openvswitch-2.3.0.tar.gz
```

下载完成后会在本地得到一个后缀为 .tar 的压缩包，将压缩包解压并进入目录 “/openvswitch-2.3.0”：

```
# tar -xzf openvswitch-2.3.0.tar.gz
# cd openvswitch-2.3.0
```

此时还不能直接执行安装脚本，因为在正式安装之前需要先编译 OVS 代码以构建交换机的内核。运行起来的 OVS 虚拟交换机实际上就是虚拟机 Linux 系统内核上的一个进程，由这个进程来完成 SDN 交换机的转发功能。通过以下命令可以完成 OVS 虚拟交换机内核的构建。

```
# make clean
# ./configure --with-linux=/lib/modules/`uname -r`/build 2>/dev/null
```

内核构建完成之后，就可以编译安装 OVS 2.3.0

```
# make
# make install
```

下面一条命令是为了让安装的 OVS 交换机支持 VLAN 功能。笔者理解的大致原理是在 OVS 源码的 openvswitch.ko 模块中有关于 VLAN 功能的支持。OVS 的开发人员可能考虑到这个功能所耗费的资源相对较大，而且并非所有使用 OVS 的研究人员都需要这个功能，所以令这个功能以一个相对独立的模块出现在 OVS 的系统架构中。

```
# modprobe gre
# insmod datapath/linux/openvswitch.ko
```

openvswitch.ko 模块完成之后，便可以安装并加载之前构建的 OVS 虚拟机内核模块。如果第一步执行失败，这一步是无法执行的。

此时 OVS 的安装过程已经完成了。但是还不能直接启动，需要使用 ovsdb

工具初始化配置数据库。笔者认为 OVS 之所以能够模拟交换机功能是因为在 OVS 进程中需要启动一个 `ovsdb_server`，这是一个轻量级数据库服务器，OVS 需要通过这个数据库服务器获取配置信息，用来配置 OVS 的内核模块。而且笔者认为这个数据库有可能会模拟存储流表或者路由表的功能，所以才能模拟交换机的转发功能。因此没有这一步骤 ovs 是完全无法使用的。

初始化配置数据库：

```
# mkdir -p /usr/local/etc/openvswitch
# ovsdb-tool create /usr/local/etc/openvswitch/conf.db
vswitchd/vswitch.ovsschema 2>/dev/null
```

接下来就可以启动 OVS 交换机。根据上面的分析，不难知道在启动 OVS 之前需要先启动配置数据库，因为 OVS 启动时要从配置数据库里面读取配置信息。

启动 `ovsdb-server` 配置数据库：

```
# ovsdb-server -v --remote=punix:/usr/local/var/run/openvswitch/db.sock
--remote=db:Open_vSwitch,Open_vSwitch,manager_options
--private-key=db:Open_vSwitch,SSL,private_key
--certificate=db:Open_vSwitch,SSL,certificate
```

初始化配置数据库：

```
# ovs-vsctl --no-wait init
```

启动 OVS 主进程：

```
# ovs-vswitchd --pidfile --detach#
```

上面说过 OVS 其实就是 Linux 系统内核空间里的一条线程，因此要查看 OVS 是否启动成功只需要查看线程是否成功启动即可，linux 的“ps”命令可以让我们方便的查看进程状态：

```
# ps -ef | grep ovs
```



```
root@zhl-virtual-machine:~/mininet# ps -ef | grep ovs
root      1369      1  0 18:17 ?        00:00:37 ovsdb-server: monitoring pid 1370 (healthy)

root      1370    1369  0 18:17 ?        00:00:01 ovsdb-server /etc/openvswitch/conf.db -vconsole:emer -vsyslog:err -vfile:info --remote=punix:/
var/run/openvswitch/db.sock --private-key=db:Open_vSwitch,SSL,private_key --certificate=db:Open_vSwitch,SSL,certificate --bootstrap-ca-cert=db:O
pen_vSwitch,SSL,ca_cert --no-chdir --log-file=/var/log/openvswitch/ovsdb-server.log --pidfile=/var/run/openvswitch/ovsdb-server.pid --detach --n
onitor

root      1409      1  0 18:17 ?        00:00:35 ovs-vswitchd: monitoring pid 1410 (healthy)

root      1410    1409  0 18:17 ?        00:00:02 ovs-vswitchd unix:/var/run/openvswitch/db.sock -vconsole:emer -vsyslog:err -vfile:info --mlock
all --no-chdir --log-file=/var/log/openvswitch/ovs-vswitchd.log --pidfile=/var/run/openvswitch/ovs-vswitchd.pid --detach --monitor
root      3185    2584  0 22:51 pts/0    00:00:00 grep --color=auto ovs
root@zhl-virtual-machine:~/mininet#
```

图 2-1 OVS 进程状态

其中 `ovsdb-server` 就是维护配置数据库的进程，`ovsdb-vswitchd` 就是 OVS 的主进程。上图所示表明 OVS 已经安装并启动成功！

### 2.2.3 sFlow-rt 流量监控环境搭建

sFlow 技术是一种以设备端口为基本单元的数据流随机采样的流量监控技术，适用于超大网络流量环境下的流量分析，方便用户实时地、详细的分析网络流量的传输情况。

sFlow-rt 是基于 sFlow 技术的流量监控工具 sFlow-rt 有 sFlow Collector 和 sFlow Agent 两部分组成。sFlow Collector 安装在一台单独的 Linux 虚拟机上，用于接受流量数据，并对得到的数据进行统计、分类、分析和展示等。sFlow Agent 安装在运行网络拓扑的虚拟机上，用于采集流量数据并上报给 sFlow Collector。一个简单的 sFlow 流量监控环境结构如图 2-2 所示，更为复杂的环境可以在此基础上扩展。

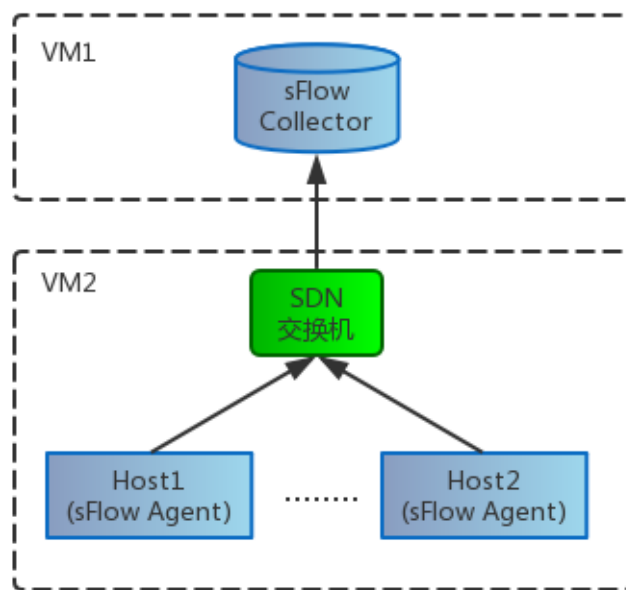


图 2-2 sFlow 拓扑实例

笔者这里以两台主机，一台交换机的网络拓扑讲解 sFlow 流量监控工具安装和部署。

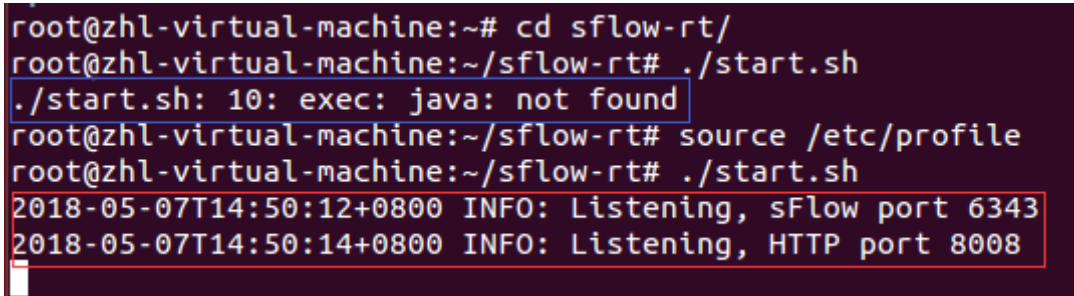
现在我们需要创建一台新的 Linux 虚拟机用来安装 sFlow Collector。新的虚拟机创建完成之后，打开 terminal 终端，开始安装步骤：

首先获取 sFlow-rt 的安装包，可以直接通过 wget 工具从 sFlow 官网下载压缩包到本地并解压：

```
# wget http://www.inmon.com/products/sFlow-RT/sflow-rt.tar.gz
# tar -xvzf sflow-rt.tar.gz
```

解压完成后,进入/sflow-rt 目录,执行脚本文件“start.sh”以启动 sFlow Collector:

```
# cd sflow-rt
# ./start.sh
```



```
root@zhl-virtual-machine:~# cd sflow-rt/
root@zhl-virtual-machine:~/sflow-rt# ./start.sh
./start.sh: 10: exec: java: not found
root@zhl-virtual-machine:~/sflow-rt# source /etc/profile
root@zhl-virtual-machine:~/sflow-rt# ./start.sh
2018-05-07T14:50:12+0800 INFO: Listening, sFlow port 6343
2018-05-07T14:50:14+0800 INFO: Listening, HTTP port 8008
```

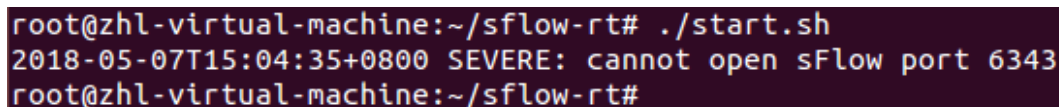
图 2-3 启动 sFlow Collector

出现如图红色方框所示内容时说明 sFlow Collector 已经启动成功,端口 6343 和端口 8008 已经处于 Listening 状态。

在启动 sFlow Collector 可能会出现如图蓝色方框所示的错误,这是因为 sFlow-rt 的运行需要 java 环境,如果虚拟机没有 Java 环境,则需要先安装 jdk。关于 Linux 下 jdk 的安装和环境配置,读者可以自行百度。多说一句,有的虚拟机即使配置好了 java 环境也会报错,在确保 jdk 已经成功安装和环境变量已经配置好的情况前,可以使用命令“source /etc/profile”对环境进行修复。

在图 2-3 所示的视图下通过“Ctrl + Z”可以停止 sFlow Collector

启动 sFlow 时可能会遇到如下错误:



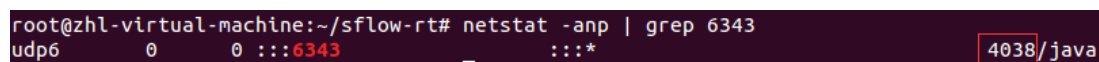
```
root@zhl-virtual-machine:~/sflow-rt# ./start.sh
2018-05-07T15:04:35+0800 SEVERE: cannot open sFlow port 6343
root@zhl-virtual-machine:~/sflow-rt#
```

图 2-4 sFlow Collector 启动失败

这是因为 6343 端口被占用,可能是 sFlow 未正常关闭,也有可能是被其他进程占用,解决方法是将占用改端口的进程杀死。

首先通过以下命令查看占用改端口的进程的 pid:

```
# netstat -anp | grep 6343
```



```
root@zhl-virtual-machine:~/sflow-rt# netstat -anp | grep 6343
udp6 0 0 :::6343 :::* 4038/java
```

图 2-5 6343 端口占用



然后通过以下命令将进程杀死后，再次启动即可：

```
# kill -9 [pid]
```

启动成功后从虚拟机浏览器访问 <http://localhost:8008/html/index.html>，可进入 sFlow web 页面：

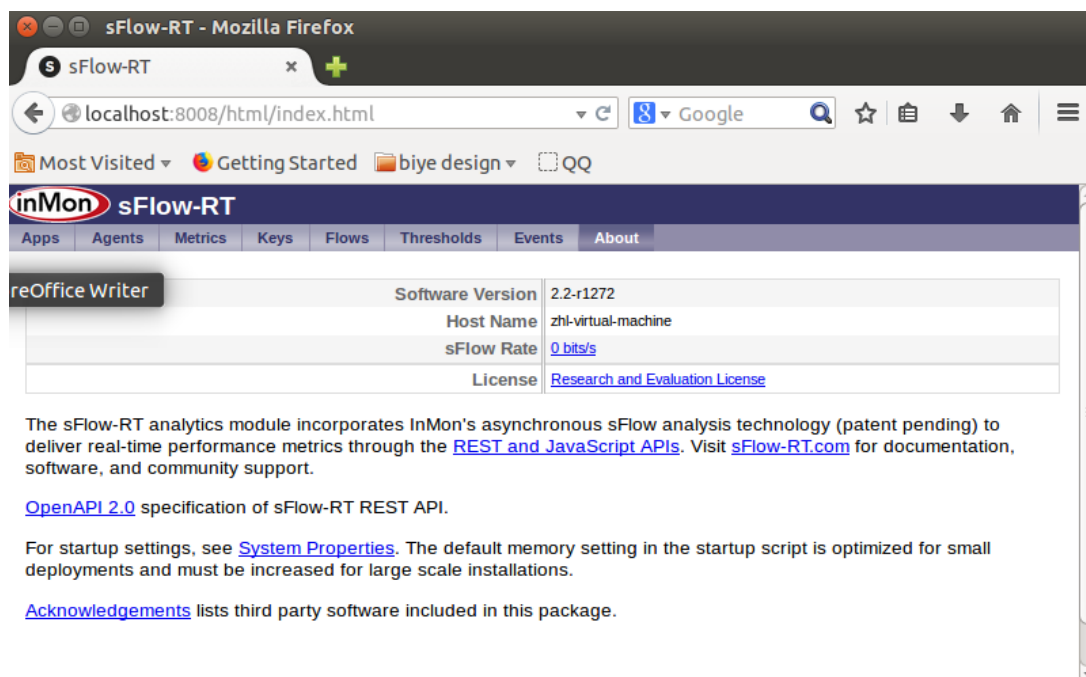


图 2-6 sFlow-RT 浏览器主页

安装完 sFlow Collector 之后，还要配套地部署 sFlow Agent，由于 OVS 和 Mininet 上面都做了对 sFlow 的支持，所以只需要开启对应的功能，并配置 sFlow Agent 即可。需要在安装 Mininet 和 OVS 的虚拟机上做以下操作。

开启 Linux 终端窗口，获取 root 权限，执行以下命令：

```
# ovs-vsctl -- --id=@sflow create sFlow agent=s1
target="\ 192.168.115.130:6343\" header=128 sampling=64
polling=1 -- set bridge s1 sflow=@sflow
```

其中 agent 是要进行流量监控的网络设备的网卡。

target 是 sFlow Collector 的地址，加粗部分为 sFlow Collector 主机的 IP，端口默认为 6343，本次课题不需要关注其他的参数，所以笔者不在详细阐述。

bridge：需要开启 sflow 的网桥。

执行上述命令，若操作成功 terminal 终端会打印一个 UUID。

```
0ede6a2a-7654-4ac5-8722-1341c2ad9448
```

还可以通过以下命令查询 sFlow Agent 的配置信息：

```
root@zhl-virtual-machine:~# ovs-vsctl list sflow
_uuid          : 0ede6a2a-7654-4ac5-8722-1341c2ad9448
agent          : "s1"
external_ids   : {}
header         : 128
polling        : 1
sampling       : 64
targets        : ["192.168.115.130:6343"]
```

这里需要注意的是在配置 sFlow Agent 之前需要现在虚拟机上建立好需要监控的虚拟网络拓扑,换句话说就是确保配置的 agent 和 bridge 是真实存在的,否则会报如下错误：

```
ovs-vsctl: no row "s1" in table Bridge
```

配置完成之后,如果 target 参数配置的 sFlow Collector 地址是真实存在的,那么在 sFlow Collector 的网页的“Agent”会看到节点的信息：

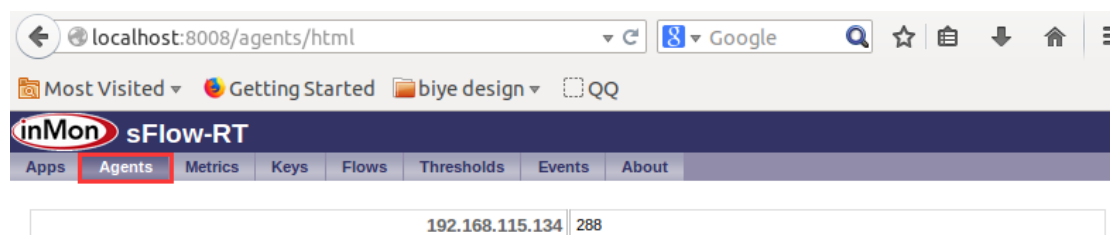


图 2-7 sFlow Agent 节点信息

至此, sFlow 流量监控环境的搭建完成。

## 第 3 章 实验目标分析和功能分析

### 3.1 实验目标分析

课题的名称为“SDN 环境下的虚拟网络搭建”，经过指导老师的讲解和知道最终确定的课题目标是，利用现有的通用网络模拟工具大家一个虚拟的软件定义网络，在确保处事网络拓扑搭建成功的前提下，对网络结构进行一些科学的、合理的改变，然后验证更改后的虚拟网络拓扑的连通性。以此来证明在一个虚拟网络中，局部某个网络节点或者网络链路的改变并不会影响剩余节点原来的连通性。

#### 3.1.1 虚拟网络拓扑规模

我们的工作是利用 SDN 的方式模拟一个虚拟网络，而不是在一个真实的物理网络上划分出若干个相互隔离的虚拟网络来。因此规模不需要媲美真实网络（上百个计算节点），但是为了课题结论的说服力着想，规模也不宜太小。经过与指导老师的讨论最终确定以 15 - 20 个网络节点为最佳，这样既不会因为网络结构的规模太小而是论文的结论丧失说服力，也不会因为网络结构太大而出现由于计算设备性能瓶颈而带来的种种问题。

而且对于我们选择的开发工具来讲，Mininet 对这种体量的胖树结构的支持完全没有问题。因此，最终确定课题实验所需的虚拟网络拓扑的网络节点的数量在 15 - 20 个，节点类型采用 Mininet 支持的 OVS 交换机。

### 3.1.2 虚拟网络拓扑结构

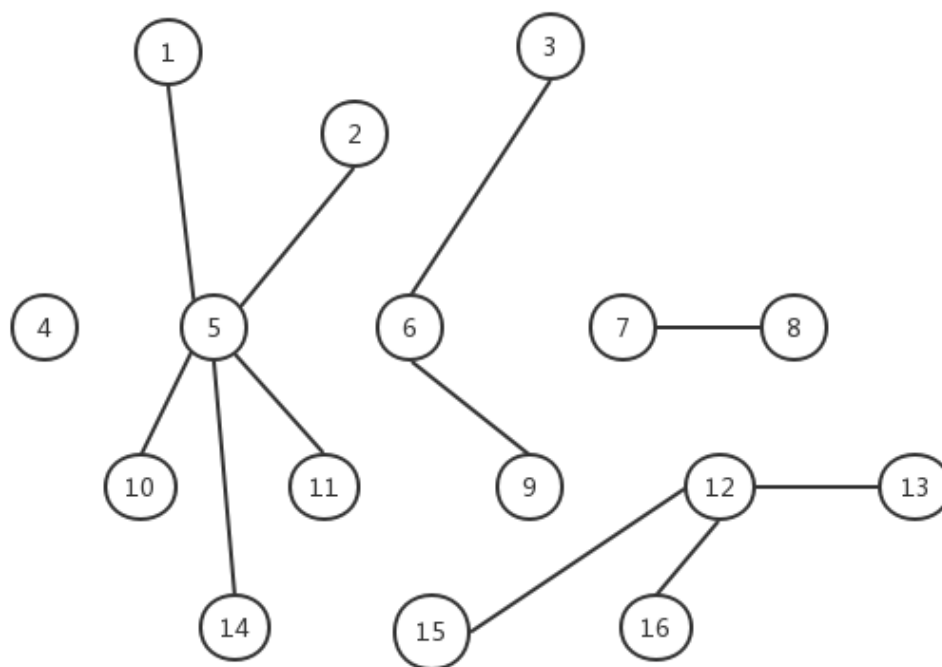


图 3-1 虚拟网络拓扑结构

这里设计的网络拓扑为 16 个网络节点的拓扑结构，满足期望的网络规模，而且整个网络结构中不等的划分为 5 个相互隔离的虚拟网络，之后的章节会通过实践证明这几个虚拟网络内部是互相连通的而且内部的流量不会外泄。各个虚拟网络之间不会彼此影响，即删除某个虚拟网络内部网络节点或者某条网络链路，不会影响其他虚拟网络的连通性，而且被修改的虚拟网络内部的流量依然不会外泄。

## 3.2 实验功能分析

### 3.2.1 虚拟网络拓扑搭建方式

之前提到过 Mininet 本身提供了一个简单的网络结构（一个虚拟机，两台主机，一个控制器节点）。在建立了这个基本结构之后，可以在 mininet>视图中通过 Mininet 提供的 CLI 命令来手动的添加节点和链路。添加一个网络节点或者一条网络链路都需要手写一条 Mininet 命令。这种方式虽然可以完成目标

拓扑结构的搭建，但是工作量比较大，而且工作成果不具备复用性，一旦退出 mininet 或者关机就要重新建立拓扑结构。在尝试了这种方式之后，笔者发现这种方式有一种致命的缺点，通过命令后期添加的主机节点不会自动分配 IP。这样就会导致即使你在两个节点之间添加了一条直接的链路或者通过一个交换机节点来连接，节点互相之间也不会 ping 通，需要你手动配置 IP。这将大大降低工作效率。

通过查阅官网发现 Mininet 提供了一套 Python API，可以根据这套 API 编写自定义拓扑的脚本，在脚本中通过 Mininet 暴露出来的接口以代码的形式搭建网络拓扑。这种方式明显比上一种方法更加通用、专业，工作量更小。根据官网给出的 API 文档可以看出，在 mininet.topo.Topo 类里暴露出来一些可供我们使用的接口，如下图：

addNode(self, name, opts)	Add Node to graph.
addSwitch(self, name, opts)	Add switch to graph.
addLink (self, node1, node2, port1 = None, port2 = None, key = None, opts)	Add a link between node1 and node2.
addHost(self, name, opts)	Add host to graph.

表 3-1 mininet.topo.Top 的部分 API 接口

再仔细分析一下，这种方式构建网络拓扑虽然工作量减少了而且不需要每次启动 Mininet 都重新建立拓扑，但是这样还是存在代码复用性低的问题，一个脚本文件只能用来创建一个特定的网络拓扑结构，如果要想实现另一个新的拓扑结构，需要重新编写一个脚本文件。

既然都能通过脚本文件来创建自定义拓扑，为什么不能写一个通用的脚本文件呢？按照这个思路，我们需要找一种能够表示虚拟网络拓扑结构的数据结构，邻接矩阵可以解决图的表示。在创建网络拓扑之前根据邻接矩阵的值，创建虚拟网络拓扑。这样每次创建不同的网络结构是只需要给出对应的邻接矩阵即可。这种方式显然比之前的两种方式更优雅。

### 3.2.2 Mininet 视图下对网络结构的二次重构

根据实验目的，我们要做的工作除了搭建虚拟网络之外，还要在当前的网络拓扑运行的情况下对拓扑结构进行一定的修改或者重构，但是不能导致原有的网络结构崩溃，也不能破坏网络拓扑的连通性。

根据对网络图谱搭建方法的探讨，在 `mininet>` 视图中通过命令来删除网络节点或者网络链路是我们不能接受的。所以我们应该也想一种方便的方法来一次性删除多个节点或者多条链路。通过查询 Mininet 官方 Python API 文档，在 `mininet.net.Mininet` 中提供了删除网络节点或者链路的接口：

<code>delLinkBetween(self, node1, node2, index = 0, allLinks = False)</code>	Delete link(s) between node1 and node2.
<code>delNode(self, node, nodes = None)</code>	Delete node.
<code>delSwitch(self, name)</code>	Delete a switch
<code>delHost(self, name)</code>	Delete a host.

图 3-2 `mininet.net.Minine` 的部分 API 接口

我们可以根据这几个接口来完成对运行中网络拓扑进行重构的目的。同样的我们可以设计一种通用的网络链路和网络节点表达方式存储在 `txt` 文件中，在重构时将 `txt` 文件作为参数传递给 `Mininet` 对象来完成拓扑结构的重构。

需要注意的是在删除链路时，可以保留节点。因为即使链路不存在，单独的网络节点（主机或交换机）不会影响网络的连通性也不会造成虚拟网络的流量外泄。因为如果该节点与多个节点相连接，删除与其中一个节点之间的网络链路时，该节点与其他网络节点的联通不受影响。与被删除链路的节点之间不存在网络链路，流量自然不会泄露。但是删除节点时，需要把与该节点相关的、可删除的链路一起删除，因为无用的链路存在是会消费网络流量的，会造成网络流量外泄。所以删除节点时，需要进行一些特殊处理，即确保无用网络链路被删除，同时确保其他链路不会被误删。

关于 `Mininet` 内部的实现原理、需要扩展的类以及具体的扩展方式在后续章节中阐述。

## 第 4 章 实验原理剖析及实现方法分析

经过前面章节的分析，本次实验的大部分工作是借助 Mininet 来完成的，而且要实现本次实验的最终目标必须对 Mininet 的功能进行定制化的扩展。所谓“工欲善其事，必先利其器”。对 Mininet 有一个清晰的、源码层次上的理解将对我们后续的开发工作大有裨益。本章将对 Mininet 的源码组织结构和核心功能代码进行详细的解读。

### 4.1 实验原理剖析

#### 4.1.1 SDN 与网络虚拟化

网络虚拟化是一种网络技术，通过网络虚拟化可以在一片真实的物理网络中划分出若干个相互隔离的虚拟网络，每个虚拟网络之间独立工作，互不影响。提高了网络资源的利用率。而 SDN 是一种新型的、创新型的网络架构，它将传统网络架构的数据平面和控制平面分离，由 SDN 网络设备完成分发，而控制命令的下发交给更上一级的控制器来完成。这种新型的网络架构大大提高了网络的灵活性和开放式网络编程能力，很大程度上解决了传统网络的业务适应能力弱，业务部署低效的问题。

利用 SDN 实现网络虚拟化需要完成三个工作，即物理网络管理、网络资源虚拟化和网络隔离，而这三部分工作一般需要通过一些虚拟化平台来完成。所谓的虚拟化平台一般是指敏捷控制器。控制器处于应用层，负责将租户定制的配置数据转换成硬件可识别的网络报文（如 netconf<sup>[13]</sup> 报文）下发给网络设备，网络设备根据报文内容来改变自己的行为。

同时虚拟化平台还要负责完成网络资源的虚拟化，而这部分内容最主要的是网络节点资源的虚拟化和网络链路资源的虚拟化两部分。所谓的网络资源虚拟化就是将物理网络的拓扑结构和网络节点抽象出来，以某种数据可视化的方式展示给租户，租户在控制器上对抽象出来的网络进行一系列操作，例如重新配置网络节点、更改拓扑结构等。然后将这些操作反映到真实的物理网络中。

在软件定义网络中可以通过 VLAN 技术来划分虚拟网络实现网络隔离，同时

还要注意不同租户之间的业务隔离。因为大多数的敏捷控制器都是面向租户的，而不同租户之间的必然有不同之处，两者缺一不可。

如果建立一个真实的物理网络，并在上面进行虚拟网络的划分试验及 SDN 技术，这种方式的成本太高。所以一般会采用方针工具来构建符合实验要求的虚拟拓扑。这样的方针工具有很多，而与我们是实验最匹配的是 Mininet 仿真工具。

#### 4.1.2 Mininet 功能逻辑

在正式开发之前，我们需要对 Mininet 的整体的功能逻辑和 workflow 有一个清晰的认识。下图所示是一个典型场景：

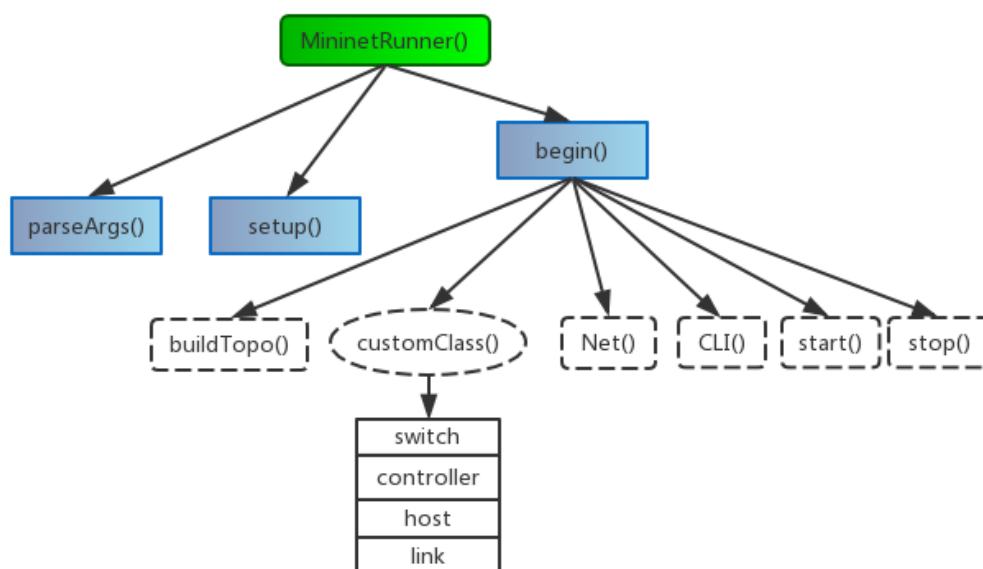


图 4-1 Mininet 功能逻辑

/mininet/bin 目录下面定义了一个 mn 文件，在启动 Mininet 时会调用这个文件。mn 文件中定义类一个 MininetRunner() 类用来处理网络拓扑初始化的核心工作。而这部分的核心就是在 MininetRunner() 中定义的三个方法：parseArgs()、setup()、begin()，大致的处理逻辑如上图所示。

首先在使用“sudo mn”启动 Mininet 时，会先读取 cli 参数并调用 parseArgs() 解析。根据用户提供的参数构建符合要求虚拟网络拓扑。例如用户在启动 Mininet 是传递了“--custom”参数，这是 Mininet 会读取指定的自定



Setup()方法主要处理 Mininet 日志相关的逻辑,对日志进行实例化和一些必要的参数配置。

首先会初始化 topo、host、switch、link 等网络资源,网络资源初始化的具体工作定义在/mininet/mininet/util.py 中。以上工作完成之后,还会先配置 CLI。最后启动网络拓扑。启动过程的写法很巧妙,Mininet 作者用一个 mn 对象(Net 类型)持有了上述的所有资源以及对象,然后通过 mn.start() 去调用它所持有的对象的 start() 方法。

图 4-2 mn 对象的初始化及启动

## 4.2 Mininet 源码解读

### 4.2.1 Mininet 源码结构分析

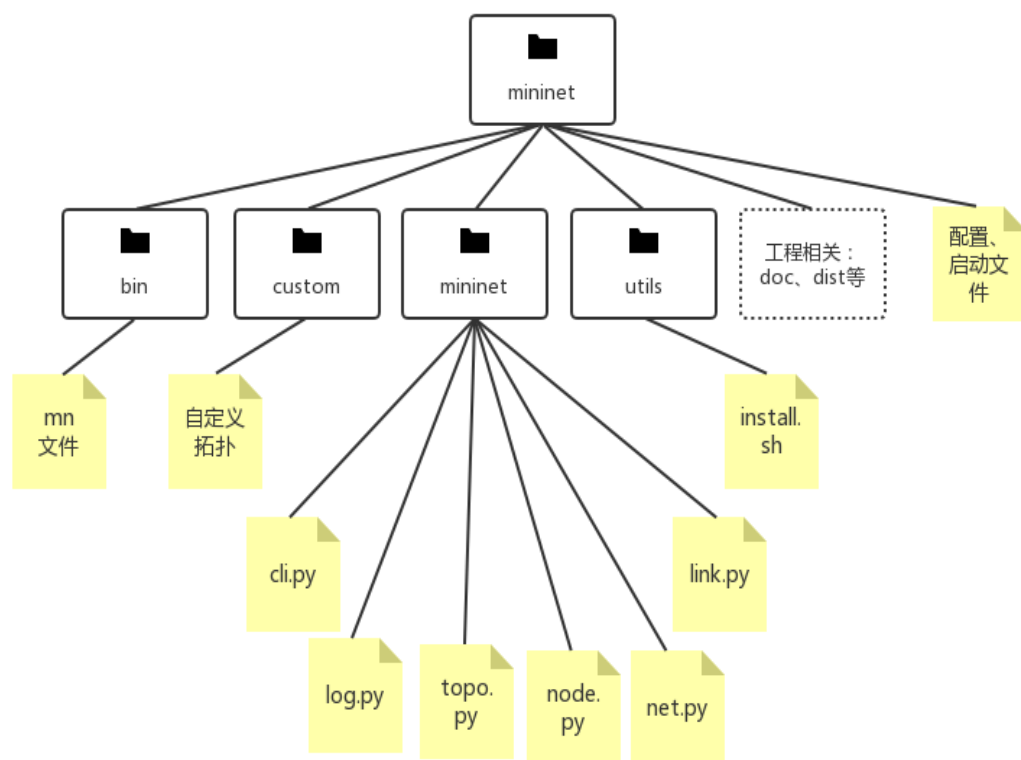


图 4-3 Mininet 源码组织结构

这里只介绍一些对我们对 Mininet 功能进行扩展用到的模块和核心模块。

我们将 Mininet2.3.0 的源码下载到本地之后，解压会得到一个 mininet 文件夹，文件夹内存放了若干个子文件夹和若干个文件（包括 .sh 文件、.py 文件等）。这些目录下的文件不需要太多关注因为基本上是一些核心的配置信息和 Mininet 版本信息，以及编译文件。除此之外 /build、/doc、/dist 文件是一些工程性的文件夹，里面的文件与我们的实验工作不相关。因此这里不再赘述

**bin 目录：**里面只有一个 mn 文件，是一个运行相关的文件，该文件里主要处理一些 Mininet 启动和网络拓扑初始化的工作。该文件源码的相关解读已在 4.1.2 节中给出。

**custom 目录：**一般自己定义的网络拓扑脚本建议放在这个目录下面。该目

录里面默认有一个“`topo-2sw-host.py`”的文件。该文件建立了一个两主机、两虚拟机的网络拓扑结构，是 Mininet 作者提供的一个自定义虚拟网络拓扑的模板。开发者可根据这个模板学习如何编写自己的网络拓扑脚本。在本次试验中，笔者将表示网络拓扑结构、待删除节点和待删除链路的 `txt` 文件也放在此目录下便于管理。

**mininet 目录:** 这个目录是整个工程的核心，里面存放了若干 python 文件，全部是搭建网络拓扑的核心组件。其中 `node.py` 文件内部定义了用来对主机、虚拟机、控制器 Controller、网桥等进行抽象化的 python 类。其中虚拟机对象通过继承关系又进一步扩展出了多个不同类型的虚拟机类，如 `UserSwitch`、`OVSSwitch`、`IVSSwitch`。Controller 又扩展出了 `Ryu`、`NOX`、`OVSController` 等。Mininet 作者将所有不同类型的网络节点都定义成 `node` 类型，将大大方便节点资源的添加和删除操作。

`link.py` 中定义了一个 `Link` 类，用来抽象两个网络节点之间的虚拟网络链路。我们接下来的网络拓扑重构工作中的删除网络链路，就是要对 `Link` 对象进行操作。在 mininet 中链路可以进行删除、添加、启用、禁用操作，还可以设置延迟，带宽等属性。

`topo.py` 其实定义了一个图，而图的结构解释网络拓扑的结构，我们添加的网络节点就相当于图中的点，网络链路就是图中的边。。内部有一个 `Topo` 类用来抽象网络拓扑，我们添加的节点和链路都被整合成了 `Topo` 对象，再用这个 `topo` 对象构造 Mininet 对象。我们自定义的网络拓扑就是通过继承 `Topo` 类来实现的。

`net.py` 是比较核心的一个文件，也是我们本次实验需要着重重写的类。在这个文件里定义了一个 `Mininet` 类，他的构造函数接受几乎所有的网络节点资源来构造最终呈现为 `mininet>s` 视图下的网络拓扑。它的内部成员与 `Topo` 类相似，都有用于只能删网络节点和网络链路的成员函数，但是比 `Topo` 多出一些进行仿真操作的成员函数，例如：`pingAll()`、`iperf()`。

`cli.py` 该定义了 `mininet>`视图下的命令行，如 `nodes`(列出所有网络节点)、`links` (列出所有网络链路)、`link node1 node2 up/down` (启用或者停用一条链路)、`pingall` (网络上的所有 Host 节点两两执行 ping 操作)、`iperf node1`

node2 (测试 node1 和 node2 之间网络链路的性能)。在 cli.py 里面定义了很多函数名为“do\_operation”的函数，这样的每个函数都对应一条命令。其对应关系在 mn 文件的 TESTS 和 ALTSPELLING 中定义。

另外在这个目录下会看到若干个后缀名为.pyc 的文件，这是 python 文件编译之后的结果。单独编译一个 python 文件不会产生.pyc 文件，但是如果一个 python 文件使用 import 语句引用了另一个 python 文件，在编译之后会生成一个与被引用文件同名的 pyc 文件。

**util 目录：**在这个目录下我们只需要关注 install.sh 即可。在 2.2.1 节中我们介绍过通过 “./install.sh -a” 可以全量安装 mininet 所有依赖。而我们在对 mininet 的功能完成扩展之后，需要见自己的代码重新编译一下才能生效。这是就要再次用到 install.sh。我们可以使用命令 “./install.sh -n” 来编译生成的代码。

#### 4.2.2 核心代码 net.py 解读

之前已经对 net.py 进行过说明，但是因为这个文件比较核心，而且与本次实验的功能性需求息息相关，所以这里结合源码详细地分析一下这个类：

```
import os
import re
import select
import signal
import random
import fileinput
from time import sleep
from itertools import chain, groupby
from math import ceil
from mininet.cli import CLI
from mininet.log import info, error, debug, output, warn
from mininet.node import ( Node, Host, OVSKernelSwitch, DefaultController,
                           Controller )
from mininet.nodelib import NAT
from mininet.link import Link, Intf
from mininet.util import ( quietRun, fixLimits, numCores, ensureRoot,
                           macColonHex, ipStr, ipParse, netParse, ipAdd,
                           waitListening )
from mininet.term import cleanUpScreens, makeTerms
# Mininet version: should be consistent with README and LICENSE
VERSION = "2.3.0d1"

class Mininet( object ):
    "Network emulation with hosts spawned in network namespaces."

    def __init__( self, topo=None, switch=OVSKernelSwitch, host=Host,
                  controller=DefaultController, link=Link, intf=Intf,
                  build=True, xterms=False, cleanup=False, ipBase='10.0.0.0/8',
                  inNamespace=False,
                  autoSetMacs=False, autoStaticArp=False, autoPinCpus=False,
                  listenPort=None, waitConnected=False ):
```

图 4-4 Mininet 类构造方法

首先是一些库文件和模块的引入，无需多做说明。再往下声明了 Mininet 的版本。紧接着声明了一个 Mininet 类，这个文件 80%的代码都是这个类的实现，这个类的构造函数的参数非常之多，几乎涵盖了一个虚拟网络需要的所有网络元素。而且每个参数都有一个默认值，所以即使开发者不传入任何参数，仅仅执行“sudo mn”命令式依然可以成功创建一个网络拓扑。

紧接着是 Mininet 一些类变量的声明和初始化，共有 26 个相关的变量，而我们只需要关注我们需要的部分就好，即：

```
self.hosts = []
self.switches = []
self.controllers = []
self.links = []
self.nameToNode = {} # name to Node (Host/Switch) objects
```

图 4-5 Mininet 部门成员变量

其中 hosts、switches、controllers、links 是分别用来存放主机、虚拟交换机、控制器、网络链路的列表。很自然的可以联想到在增加或删除网络节点时会修改这些列表持有的对象。但这只是其中一步操作而已。其中 nameToNode 是一个字典，根据注释可以知道这个字典同时存储主机、虚拟交换机两类节点。以节点的 name 为 key，value 自然是节点对象。当无法知道某个节点在对应列表中的具体位置时，这个字典就可以大显身手了。

再往下定义了一些常规的添加、删除网络节点 add\*\*\*\*() 和 del\*\*\*\*() 方法，逻辑也比较简单清晰，不再赘述。这里介绍一些 Mininet 类的高级特性。

```
def startTerms( self ):
    "Start a terminal for each node."
    if 'DISPLAY' not in os.environ:
        error( "Error starting terms: Cannot connect to display\n" )
        return
    info( "*** Running terms on %s\n" % os.environ[ 'DISPLAY' ] )
    cleanUpScreens()
    self.terms += makeTerms( self.controllers, 'controller' )
    self.terms += makeTerms( self.switches, 'switch' )
    self.terms += makeTerms( self.hosts, 'host' )
```

图 4-6 Mininet.startTerms() 方法

当执行 cli 命令“xterm node1”时会执行此方法，打开虚拟网络设备 node1 的终端，这样就可以对某个网络设备单独进行某些配置，在虚拟网络拓扑运行

的过程中改变某台设备的行为，这个完全称得上对虚拟网络的高度仿真。这也是 mininet 的强大之处。

```
def ping( self, hosts=None, timeout=None ):
    packets = 0
    lost = 0
    ploss = None
    if not hosts:
        hosts = self.hosts
        output( '*** Ping: testing ping reachability\n' )
    for node in hosts:
        output( '%s -> ' % node.name )
        for dest in hosts:
            if node != dest:
                opts = ''
                if timeout:
                    opts = '-W %s' % timeout
                if dest.intfs:
                    result = node.cmd( 'ping -c1 %s %s' % (opts, dest.IP()) )
                    sent, received = self._parsePing( result )
                else:
                    sent, received = 0, 0
                packets += sent
                if received > sent:
                    error( '*** Error: received too many packets' )
                    error( '%s' % result )
                    node.cmdPrint( 'route' )
                    exit( 1 )
                lost += sent - received
            output( ( '%s ' % dest.name ) if received else 'X ' )
        output( '\n' )
```

图 4-7 Mininet.ping()方法

该方法用于执行主机间的 ping 操作，既支持两台主机之间互相 ping，也支持所有主机节点之间两两互 ping。在 Mininet 中定义了一个 PingAll() 方法，当在 mininet>视图下执行“pingall”命令时，会调用 pingAll() 完成所有主机节点之间两两互 ping。

```
def iperf( self, hosts=None, l4Type='TCP', udpBw='10M', fmt=None,
           seconds=5, port=5001):
```

图 4-8 Mininet.iperf()方法

iperf 是用来测试网络性能的，Mininet 提供了 iperf node1 node1 命令来测试虚拟网络的性能。该命令就是调用了该方法，该方法的方法体比较庞大，因此不在本文中展示。这也是实验要求之一。

## 4.3 核心功能实现

### 4.3.1 网络拓扑初始化

在 3.2.1 节中，我们已经分析过网络拓扑的搭建方式，即将要搭建的网络拓扑的结构以邻接矩阵的形式表示，并输出一个 topo.txt 文件放在 /mininet/custom 目录下面。然后在自定义的拓扑脚本中读取邻接矩阵的值，并构造相应结构的虚拟网络拓扑。

将图 3-1 所示的虚拟网络结构转换成邻接矩阵表示如下：

```
0000100000000000
0000100000000000
0000010000000000
0000000000000000
1100000001100100
0010000010000000
0000000100000000
0000001000000000
0000010000000000
0000100000000000
0000100000000000
0000000000001011
0000000000010000
0000100000000000
0000000000010000
0000000000010000
0000000000010000
```

图 4-9 网络拓扑的邻接矩阵表示

首先要做的是要讲这个邻接矩阵读取出来并存储在一个二维数组里面，通过 Python 的 fileinput 模块可以完成文件读取功能

```
def initTopo(filePath):
    #calculate the lineNumber of text
    res = []
    for line in fileinput.input(filePath):
        #remove the line-break
        line = line.replace("\r", "")
        line = line.replace("\n", "")
        temp = []
        for char in line:
            #transform string into integer
            temp.append(int(char))
```

```

        res.append(temp)
    print(res)
    return res

```

图 4-10 二维数组构造

initTopo() 方法接受一个 string 类型的参数，该参数为 topo.txt 的存放路径。读取操作完成之后，要根据文件的内容构造邻接矩阵。在 python 中数组内的元素是不允许更改的，所以这里采用 list 来模拟一个二维数组，同时为了方便之后的处理，矩阵的每一个元素都存储为 integer 类型。fileinput 是 python 封装好的一个文件读取模块，利用 fileinput 可以方便地对文件进行以行为单位的遍历、迭代等操作。而且该模块内部已经处理好了文件的打开方式、流的开启与关闭以及资源的释放等问题。

在 4.2.1 节的 Mininet 源码结构解读中笔者提到过，要建立自定义的网络拓扑需要继承 Topo 类，然后编写构造逻辑，既然我们已经拿到了邻接矩阵，接下来的工作解释如何在虚拟网络拓扑里面添加网络节点和网络资源了。首先要做的就是重写 Topo 的构造函数：

```

def __init__( self ):
    "Create custom topo."
    # Initialize topology
    Topo.__init__( self )
    topo = initTopo('/home/zhl/mininet/custom/topo.txt')
    length = len(topo)
    history = initHistory(length)
    #arrayList of host
    switches = []
    hosts = []
    for num in range(length):
        switches.append(self.addSwitch('s' + str(num + 1)))
        hosts.append(self.addHost('h' + str(num + 1)))
    for num in range(length):
        self.addLink(hosts[num], switches[num])
    i = 0
    for horizontal in topo:
        j = 0
        for each in horizontal:
            if each == 1:
                if(history[i][j] == 0):
                    self.addLink(switches[i], switches[j])
                    history[i][j] = 1

```



```

        history[j][i] = 1
        j = j + 1
        i = i + 1
topos = { 'mytopo': ( lambda: MyTopo() ) }

```

图 4-11 初始化网络拓扑

在构造网络拓扑主要逻辑是每个交换机节点都默认链接一个主机节点，命名格式分别为“ ‘s’ +num ”和“ ‘h’ +num ”。添加主机节点的目的是利用主机之间的 ping 操作来反应交换机的链接关系，因为在分析 Mininet 的源码时我们发现 ping 操作针对的是 Host 节点而不是 Switch 节点。然后因为 mininet 的 link 是双向的，即“node1 -> node2”建立之后再添加“node2 -> node1”或报错，因此这里定义了一个 history 二维数组来规避这种情况。

最后一句必须要有，不然无法通过命令行创建网络拓扑。至此，网络拓扑的搭建工作就完成了，若果有新的网络结构的需求，只需要修改 topos.txt 里面的内容就行即可。

### 4.3.2 批量删除网络链路

收到构建虚拟网络拓扑的启发，笔者决定删除网络链路的操作也改成类似的读取文件的方式。现在一个新的问题出现了，要多一个运行中的网络拓扑进行造作需要在 mininet>试图先进行操作，而这样怎么才能达到读取文件并删除指定网络链路的目的呢？

通过多方查阅资料，笔者发现了一种巧妙地实现方法。

Mininet 提供了很多等装好的内部命令可供开发者在 mininet>视图下对运行中的虚拟网络拓扑进行配置或者行为的更改。其中有一条“py”命令，在 mininet 运行环境下执行 Python 代码。凡是跟在“py”命令后面的语句都会每当做 python 语句运行，例如

```

mininet> py 'helloworld'
helloworld

```

而且我们在 4.2.2 中分析过，最终的网络拓扑有一个 Mininet 对象持有，“py”命令可以将 mininet>视图切换为 python 运行时环境，那么应该可以利用这个命令拿到这个 Mininet 对象。已经过笔者测试上面的思路是可行的。但是，通过在虚拟网络拓扑的运行时环境中编写大量的代码来删除指定的网络链

路的方式显然并不优雅，我们可以对 net.py 中的 Mininet 类进行扩展，然后将在自己的代码安装到 Mininet 中。这样在 mininet>视图下只需要通过“py”命令调用对应的方法即可。

根据上面的分析，我们需要通过文件读取的方式来获取要删除的网络链路，所以我们要先规定文件内容的格式。为了兼顾文件的易读性，笔者将链路的表示方式统一为“(node1,node2)”，每条链路占一行，示例如下：

```
(s1,s5)
(s6,s9)
(s12,s13)
(s5,s14)
(s7,s8)
```

下面结合代码介绍具体的实现思路。

第一步还是读取文件，并构造一个二维数组，数组的维度是  $n * 2$ ，每一个内部数组长度为 2，存放一条网络链路两端的网络节点。

```
def getLinksToDelete(self, filePath):
    links = []
    info('***links to be deleted' + '\n')
    for line in fileinput.input(filePath):
        info(line)
        #remove the line-break(compatible OS: Windows and Linux)
        line = line.replace("\n", "")
        line = line.replace("\r", "")
        #remove the '(' and ')' at the head and tail
        line = line.replace("(", "")
        line = line.replace(")", "")
        nodes = line.split(',')
        links.append(nodes)
    return links
```

图 4-12 构造待删除链路

这里需要注意的是第七行和第八行的处理。Python 的 fileinput 模块在读取文件时不会自动去掉换行符，而在后续的业务逻辑中需要手动去掉。Linux 系统中换行符为“\r\n”，不同于 windows 中的“\n”。经过上面的处理我们就可以拿到一个存储着待删除网络链路信息的二维数组。

接下来介绍删除网络链路的具体逻辑：

```
def selfDeleteLinks(self, linksFilePath):
    info('***delete serval links in one time' + '\n')
    info('***links file position : ' + linksFilePath + '\n')
```

```
links = self.getLinksToDelete(linksFilePath)
for element in links:
    node1 = self.nameToNode[element[0]]
    node2 = self.nameToNode[element[1]]
    if(self.linksBetween(node1 , node2)):
        self.delLinkBetween(node1 , node2)
```

图 4-13 删除网络链路

这里并没有什么太难的实现，只需要注意在删除链路之前判断一下要删除的链路是否存在，Mininet 类提供了 `linksBetween(node1, node2)` 方法来获取指定的链路，若不存在则返回空。

一个交互友好的程序需要给使用者适当的提示，但是为了排版的美观，笔者删除了提示代码，只留下了核心的业务逻辑处理代码。要在 `minine>` 视图中输出提示信息不能直接用 Python 的 “`print`” 语句，应该用 Mininet 提供的 “`info()`” 或者 “`output()`”。

至此，网络链路的删除功能实现完毕。

### 4.3.3 批量删除网络节点

实验的另一个要求是要删除网络节点，同样的我们还是以读取文件的方式来获取需要删除的节点信息，为了兼顾文件的可读性，笔者将节点信息的格式规定如下：

```
s2, s6, s8, s11, s10, s13
```

节点的格式就比较简单了，只是将要删除的网络节点的名称以逗号隔开，存储在 `links.txt`。网络节点的名称在创建虚拟网络拓扑的时候制定。本次实验中的 Host 节点的名称都是类似 “`h0, h1...`”，Switch 节点的名称都是类似 “`s0, s1...`”。因为读取节点文件并解析的过程很简单，所以这不在过多讲述。

在删除网络节点时需要注意的是删除节点之前，要先找出与待删除节点相关的网络链路。原因是如果只删除网络节点可能会导致存在只有一个端点的网络链路，这样会导致虚拟网络内的流量外泄。

```
def selfDeleteNodes(self, nodesFilePath):
    nodes = self.getNodesToDelete(nodesFilePath)
    #delete link related to the node(that to be deleted)
    for node in nodes:
        for element in self.switches:
```

```

pos = list(self.nameToNode.values()).index(element)
elementName = list(self.nameToNode.keys())[pos]
tempLink = self.linksBetween(self.nameToNode[node] , element)
if(tempLink):
    self.delLinkBetween(self.nameToNode[node] , element)
else:
    pass
for node in nodes:
    hostName = node.replace('s','h')
    self.delLinkBetween(self.nameToNode[node],self.nameToNode[hostName])
self.delNode(self.nameToNode[hostName])
self.delNode(self.nameToNode[node])

```

图 4-14 删除网络节点

再删除交换机节点之前，先遍历虚拟网络拓扑内的所有交换机节点，并判断每个节点与待删除节点之间是否存在网络链路。有，则删除。

这个工作完成之后还是不能删除网络节点。在 4.2.1 节中我们已经说过在创建虚拟网络拓扑时每一个交换机节点都默认链接了一个主机节点用于测试连通性和流量性能。所以再删除交换机节点的时候还需要删除对应的主机节点和两者之间的网络链路。在这些工作完成之后才能删除网络节点。同样的代码片中移除了打印提示信息的语句。

删除网络节点是最后一步的工作！

#### 4.3.4 网络性能测试及流量发送

评价网络性能的指标有很多，例如带宽质量、延迟、带宽时延积等，我们这里采用带宽质量指标来测试一下我们搭建的虚拟网络拓扑的性能，同时验证带宽质量的同时，还可以通过发送流量的方式来验证网络的连通性，比 ping 操作更有说服力。接下来面临的问题就是如何来用 Mininet 测试虚拟网络的带宽质量。

常用的网络性能测试工具有 Iperf，它可以测试 TCP 和 UDP 的带宽质量，并报告网络的延迟等信息。通过多方查阅资料发现 Mininet 内部实现了 iperf 功能而且就定义在我们需要扩展的 net.py 文件中在查看了 iperf() 方法之后发现他只能同时测试一条链路的带宽质量，显然不符合我们的要求。不过我们可以利用 Mininet 现有的方法来实现我们需要的功能。

```
def iperfAll(self , l4Type='TCP', udpBw='10M', fmt=None):
    length = len(self.switches)
    for i in range(length):
        for j in range(length - i):
            target = i + j
            if target == i:
                pass
            else:
                nodeLeft = self.switches[i]
                nodeRight = self.switches[target]
                if self.linksBetween(nodeLeft , nodeRight):
                    res = self.iperfWithoutInfo
                        ([self.hosts[i],self.hosts[target]])
                output(res)
            output('\n')
```

图 4-15 性能测试代码

iperfAll() 是笔者扩展的方法参数 l4Type 默认为 TCP，这个方法的主要内容就是对网络中的所有交换机节点之间存在网络链路进行一次性能测试。实现这个功能需要调用 iperfWithoutInfo() 方法，这个方法是笔者根据 net.py 写好的 iperf() 方法改写的，区别是该方法没有任何的输出信息，最后的测试结构封装成在一个 res 数组中返回，我们可以利用这个返回结果来组织自己的提示信息（提示信息相关代码已移除）。

#### 4.3.5 Mininet 命令扩展

虽然在 mininet>视图下通过“py”命令来调用我们在 Mininet 类中扩展的方法比直接在 mininet>视图下编写代码更方便，但是还有没有更为优雅的方式呢？

我们知道 Mininet 提供了很多 cli 命令供开发者使用，通过这些命令可以轻易的在 mininet>视图下完成对虚拟网络拓扑结构或者行为的更改，其原理大概就是对输入的命令进行解析之后通知 mininet 执行相应的方法逻辑。既然 Mininet 本身可以提供 cli 命令，那我们是不是也可以通过对 Mininet 扩展来编写我们自己的命令呢？答案是可行的，但是在动手编写代码之前我们需要搞清楚怎么来自定义自己的命令。

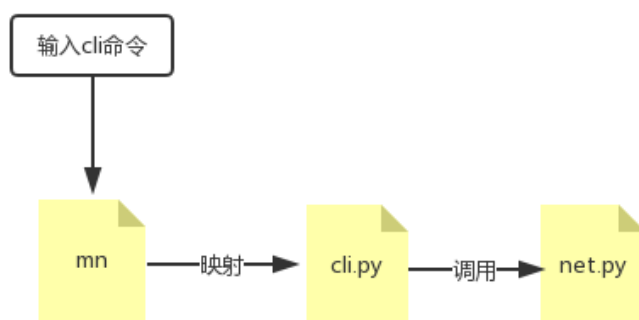


图 4-16 cli 执行过程

第一步要在 net.py 中扩展功能函数，这部分仍然是最关键的代码，因为最后的命令还是要调用这一部分的代码，我们在之前已经编写完成了。功能代码编写完之后需要扩展 cli.py 的内容。在这个文件中定义了大量的“do\_operation”函数，在 mininet>视图中执行的命令先映射到这个文件里对应的 do\_operation 函数，然后从这个函数里调用 Mininet 类里的功能代码。

```

def do_iperfall(self, line):
    args = line.split(',')
    self.mn.iperfAll()
def do_delnodes(self, line):
    if line is '':
        output('invalid arguments!' + '\n')
    else:
        self.mn.selfDeleteNodes(line)
def do_dellinks(self, line):
    if line is '':
        output('invalid arguments!' + '\n')
    else:
        self.mn.selfDeleteLinks(line)
    
```

图 4-17 cli 映射函数

上图所示代码块为笔者自己添加的三个 do\_operation 函数，依次为条用了性能测试、网络节点删除和网络链路删除三部分的功能代码。每个方法接受两个参数，第一个是固定参数，第二个 line 就是输入 cli 命令时跟在命令后面的参数，对于多参数的处理也要放在这一部分。

do\_operation 函数编写完后，需要建立 cli 命令与函数之间的映射，这部分工作要在 mn 文件中完成。mn 文件中定义了一个存放 cli 命令的列表：TESTS，

和一个存储 cli 命令与 do\_operation 函数映射关系的字典：ALTSPELLING。我们的工作是在这两个数据结构中添加自己命令和映射关系。

```
TESTS = { name: True
          for name in ( 'pingpair', 'iperf', 'iperfudp', 'pingall',
                        'iperfall', 'delnodes', 'dellinks' ) }

# Map to alternate spellings of Mininet() methods
ALTSPELLING = {'pingall': 'pingAll', 'pingpair':
               'pingPair', 'iperfudp': 'iperfUdp', 'iperfall': 'iperfall',
               'delnodes': 'delnodes', 'dellinks': 'dellinks' }
```

图 4-18 cli 映射关系

完成上述的步骤之后，cli 命令扩展的工作就算完成了！最后把虚拟机的工作目录切换到/mininet/util，利用命令“./install.sh -n”执行安装脚本，将自己的代买安装到 Mininet 中。

## 第 5 章 实验方法与结果分析

我们已经把所有的准备、开发工作完成了，接下来的工作是测试我们的程序，并利用自己搭建的实验环境对我们的观点进行验证。我们需要建立符合条件的网络拓扑，然后对网络拓扑的结构进行二次重构，然后分析实验结果，得出结论。

### 5.1 虚拟网络拓扑创建

我们首先需要来建立一个一定规模的虚拟网络拓扑，我们就按照 3.1.2 节中图 3-1 所示的结构来搭建我们的测试网络环境。先把我们之前编写的 mytopo.py 放到/mininet/custom 目录下，并将图 4-9 所示的邻接矩阵写入 topo.txt，也放入/mininet/custom 目录下。

然后打开虚拟机的 terminal 终端，获取 root 权限，讲工作目录切换到 /mininet/custom 目录：

```
zhl@zhl-virtual-machine:~$ sudo -s
[sudo] password for ***:
root@zhl-virtual-machine:~# cd mininet/custom/
root@zhl-virtual-machine:~/mininet/custom#
```

上述工作完成之后，就可以利用启动 Mininet，创建虚拟网络拓扑了，执行以下命令：

```
sudo mn --custom mytopo.py --topo mytopo --mac
```

mn 是启动文件，--custom 后面跟的参数是自定义网络拓扑脚本的文件名。

--topo 网络拓扑的结构，例如：--topo=single,5 表示创建 1 个交换机、5 个主机的拓扑结构；--topo=linear,3 表示创建一个 3 交换机、3 主机的拓扑结构，交换机之间有链路连接。

--mac 加上这个参数之后，Mininet 会为虚拟网络拓扑中的网络设备初始化 MAC 地址。

如果需要给链路设置带宽可以在上述命令后面继续添加参数：“--link tc,bw=10”，将虚拟网络内的链路带宽设置为 10Mbit，也可以在代码中设置。



执行完上述命令之后，结果如下说明 Mininet 启动成功。从图中还可以看到虚拟网络中建立的网络节点、网络链路信息，除此之外 Mininet 还自动创建了一个控制器节点。

```
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (h4, s4) (h5, s5) (h6, s6) (h7, s7) (h8, s8) (h9, s9)
(h10, s10) (h11, s11) (h12, s12) (h13, s13) (h14, s14) (h15, s15) (h16, s16) (s1,
s5) (s2, s5) (s3, s6) (s5, s10) (s5, s11) (s5, s14) (s6, s9) (s7, s8) (s12, s13)
(s12, s15) (s12, s16)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Starting controller
c0
*** Starting 16 switches
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 ...
*** Starting CLI:
```

图 5-1 启动 Mininet

启动成功之后就可以利用 links, nodes 等命令来查看虚拟网络中的网络节点和网络链路。然后通过 pingall 命令来检验虚拟网络中网络节点的连接情况是否与图 3-1 相符。

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X X h5 X X X X h10 h11 X X h14 X X
h2 -> h1 X X h5 X X X X h10 h11 X X h14 X X
h3 -> X X X X h6 X X h9 X X X X X X X
h4 -> X X X X X X X X X X X X X X X
h5 -> h1 h2 X X X X X X h10 h11 X X h14 X X
h6 -> X X h3 X X X X h9 X X X X X X X
h7 -> X X X X X X h8 X X X X X X X X
h8 -> X X X X X X h7 X X X X X X X X
h9 -> X X h3 X X h6 X X X X X X X X X
h10 -> h1 h2 X X h5 X X X X h11 X X h14 X X
h11 -> h1 h2 X X h5 X X X X h10 X X h14 X X
h12 -> X X X X X X X X X X X X h13 X h15 h16
h13 -> X X X X X X X X X X X X h12 X h15 h16
h14 -> h1 h2 X X h5 X X X X h10 h11 X X X X
h15 -> X X X X X X X X X X X X h12 h13 X h16
h16 -> X X X X X X X X X X X X h12 h13 X h15
*** Results: 79% dropped (50/240 received)
```

图 5-2 pingall 测试结果

从图 5-2 可以看出虚拟网络的通路情况与预期结果一致，说明我们对 Mininet 的扩展是正确的。但是通过 pingall 命令查看虚拟网络的通路情况只

能确保链路联通的，但链路是否能够发送流量不得而知。可以通过 iperf 间接的验证网络联络是否可以发送流量。iperf 命令只能一次验证一条网络链路，效率太低。所以我们用自定义的 iperfall 命令来发送流量：

```
mininet> iperfall
*** Iperf: testing TCP bandwidth
*** TCP bandwidth between h1 and h5:['9.56 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h2 and h5:['9.56 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h3 and h6:['9.55 Mbits/sec', '10.6 Mbits/sec']
*** TCP bandwidth between h5 and h10:['9.56 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h5 and h11:['9.56 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h5 and h14:['9.57 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h6 and h9:['9.56 Mbits/sec', '10.5 Mbits/sec']
*** TCP bandwidth between h7 and h8:['9.55 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h12 and h13:['9.56 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h12 and h15:['9.55 Mbits/sec', '10.6 Mbits/sec']
*** TCP bandwidth between h12 and h16:['9.55 Mbits/sec', '10.8 Mbits/sec']
```

## 5.2 虚拟网络拓扑重构

虚拟网络的重构主要有两种方式，一是删除若干网络链路，而是删除若干网络节点，我们的实验主要是删除虚拟交换机节点。

实验的大致思路是：利用我们自己扩展的功能删除网络节点或者网络链路后，用自己定义的 iperfall 或者原有的 pingall 查看虚拟网络的通路，验证无关的网络节点或网络链路是否被影响。

### 5.2.1 网络链路删除测试

要删除虚拟网络拓扑中的某些网络链路，可以用我们自定义的 cli 命令 “dellinks [filepath]” 来完成，首先将要删除的链路按照 4.3.2 节中规定的格式存储在文件 links.txt 中，同样将该文件放到 /mininet/custom 目录下，在 Mininet 已经启动的情况下，在 mininet>视图下执行自定义的 “dellinks” 命令，注意要把 links.txt 的路径以参数的形式传递给 mininet。最后通过提示信息查看删除的结果：

```
mininet> dellinks /home/zhl/mininet/custom/links.txt
*** delete serval links in one time
*** links file position : /home/zhl/mininet/custom/links.txt
```

```
*** links to be deleted
*** link (s1,s5) delete finished!
*** link (s6,s9) delete finished!
*** link (s12,s13) delete finished!
*** link (s5,s14) delete finished!
*** link (s7,s8) delete finished!
```

在 mininet>视图中出现上述提示信息，说明删除操作执行成功。为了确保万无一失。我们还可以通过 Mininet 自有的 cli 命令“links”查看虚拟网络拓扑的链路情况。经验证之后，指定的网络链路确实已经成功删除了，这里不再贴图。

节点删除成功后还要通过 pingall 命令来查看虚拟网络的通路情况：

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X X X X X X X X X X X X X X
h2 -> X X X h5 X X X X h10 h11 X X X X X
h3 -> X X X X h6 X X X X X X X X X X
h4 -> X X X X X X X X X X X X X X X
h5 -> X h2 X X X X X X h10 h11 X X X X X
h6 -> X X h3 X X X X X X X X X X X X
h7 -> X X X X X X X X X X X X X X X
h8 -> X X X X X X X X X X X X X X X
h9 -> X X X X X X X X X X X X X X X
h10 -> X h2 X X h5 X X X X h11 X X X X X
h11 -> X h2 X X h5 X X X X h10 X X X X X
h12 -> X X X X X X X X X X X X X h15 h16
h13 -> X X X X X X X X X X X X X X X
h14 -> X X X X X X X X X X X X X X X
h15 -> X X X X X X X X X X X h12 X X h16
h16 -> X X X X X X X X X X X h12 X X h15
*** Results: 91% dropped (20/240 received)
```

图 5-3 删除链路后 pingall

接下来根据 pingall 的结果来分析是否符合预期。

首先，去掉网络链路（s1<->s5）、（s5<->s14）之后，主机 h1、主机 h14 与其他主机节点都不存在通路，h2、h10、h11 与 h1、h14 不再互相连通，h5 与 h14 不再互通。

然后，去掉网络链路（s6<->s9）之后，h3、h6 与 h9 不再互通。

去掉网络链路（s7<->s8）之后，h7、h8 有其他主机节点都不在联通。

最后，去掉网络链路（s12<->s13）之后，主机 h13 与其他主机节点都不在联通。

综上所述，图 5-3 所示的结果与以上分析完全相符。然后通过 iperfall 查看剩余网络链路是否真实可用。

```
mininet> iperfall
*** Iperf: testing TCP bandwidth
*** TCP bandwidth between h2 and h5:['9.56 Mbits/sec', '10.3 Mbits/sec']
*** TCP bandwidth between h3 and h6:['9.57 Mbits/sec', '10.9 Mbits/sec']
*** TCP bandwidth between h5 and h10:['9.56 Mbits/sec', '10.7 Mbits/sec']
*** TCP bandwidth between h5 and h11:['9.55 Mbits/sec', '10.6 Mbits/sec']
*** TCP bandwidth between h12 and h15:['9.56 Mbits/sec', '10.9 Mbits/sec']
*** TCP bandwidth between h12 and h16:['9.56 Mbits/sec', '10.7 Mbits/sec']
```

根据流量的检测结果，剩余网络链路依然联通，且性能未受影响。

### 5.2.2 网络节点删除测试

要删除虚拟网络拓扑中的某些网络节点，可以用我们自定义的 cli 命令“delnodes [filepath]”来完成，首先将要删除的链路按照 4.3.3 节中规定的格式存储在文件 nodes.txt 中，同样将该文件放到/mininet/custom 目录下，在 Mininet 已经启动的情况下，在 mininet>视图下执行自定义的“delnodes”命令，注意要把 nodes.txt 的路径以参数的形式传递给 mininet。最后通过提示信息查看删除的结果：

```
mininet> delnodes /home/zhl/mininet/custom/nodes.txt
s2, s6, s8, s11, s10, s13
```

在 mininet>视图出现以上提示信息，说明删除网络节点成功，而且与节点相关的链路也被删除。可以通过 nodes 和 links 命令确认删除是否成功。

```
mininet> nodes
available nodes are:
c0 h1 h12 h14 h15 h16 h3 h4 h5 h7 h9 s1 s12 s14 s15 s16 s3 s4 s5 s7 s9
```

可见指定的交换机节点已经成功删除，对应的主机节点也被删除。之后通过 pingall 命令查看网络通路情况：

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X h5 X X X h14 X X
h3 -> X X X X X X X X X
h4 -> X X X X X X X X X
h5 -> h1 X X X X X h14 X X
h7 -> X X X X X X X X X
h9 -> X X X X X X X X X
h12 -> X X X X X X X h15 h16
h14 -> h1 X X h5 X X X X X
h15 -> X X X X X X X h12 X h16
h16 -> X X X X X X h12 X h15
*** Results: 86% dropped (12/90 received)
```

图 5-3 删除节点后 pingall

删除交换机节点 s2, s6, s8, s11, s10, s13 之后，网络链路 (s2<->s5)、(s6<->s3)、(s6<->s9)、(s8<->s7)、(s11<->s5)、(s10<->s5)、(s13<->s12) 也会被删除。然后主机节点 h3、h4、h7、h9 与其他所有节点都不互通，主机 h1、h5、h14 相互连通，主机 h12、h15、h16 相互连通。

图 5-3 显示的结果与上述分析完全相符。

然后再通过 iperfall 命令来测试连通的网络链路是否可以发送流量。

```
mininet> iperfall
*** Iperf: testing TCP bandwidth
*** TCP bandwidth between h1 and h5:['9.56 Mbits/sec', '10.8 Mbits/sec']
*** TCP bandwidth between h5 and h14:['9.56 Mbits/sec', '10.8 Mbits/sec']
*** TCP bandwidth between h12 and h15:['9.56 Mbits/sec', '10.8 Mbits/sec']
*** TCP bandwidth between h12 and h16:['9.55 Mbits/sec', '10.8 Mbits/sec']
```

如图所示，其余网络链路可以正常发送流量，且网络性能未受明显影响。

一组实验数据无法得出有说服力的结论，这里因为篇幅的限制只是简单的以其中一组实验数据为例来介绍我们的试验方法和分析的思路。

## 第 6 章 结论

SDN 环境下搭建的虚拟网络与传统的从真实物理网络中隔离出来的虚拟网络拥有相同的功能特点。虚拟网络内部的网络节点是互相连通的而且内部的流量不会外泄。除此之外，虚拟软件定义网络支持利用开放的网络可编程技术动态的更改虚拟网络的结构和行为。在以删除若干网络节点和网络链路的方式更改虚拟网络结构时，不会对不相关的虚拟网络的连通性和性能造成影响，而且流量依然不会外泄。

我们在第五章中已经做了严格的实验验证。首先我们构建了一个有 16 个虚拟交换机节点构成的虚拟网络拓扑，网络拓扑内的链路分布情况如图 3-1 所示。因为 Mininet 未提供交换机节点中之间的 ping 操作接口，所以我们给每个虚拟交换机都默认连接了一个主机节点用来直观的展示虚拟网络的通路情况。

网络拓扑搭建完成之后，我们首先进行了网络链路的删除测试。通过删除一些指定的网络链路改变虚拟网络拓扑的整体结构。通过对网络拓扑内剩余节点之间的连通情况进行验证，我们发现实验结果和我们理论分析的结果一致。并通过 Mininet 提供的 iperf 工具给虚拟网络拓扑内的网络链路发送流量，证明当前网络性能并未受被删除链路的影响。

然后将网络拓扑恢复到删除链路之前的结构，进行了网络节点的删除测试，通过对指定网络节点和相关的网络链路的删除操作改变虚拟网络拓扑的结构。并通过网络拓扑连通性验证，和给网络拓扑内的网络链路发送流量，返现实验结果和我们理论分析的结果完全一致。证明当前网络性能并未受被删除链路的影响。

最终我们通过实验得到结论：通过 SDN 方式搭建的虚拟网络与传统定义的虚拟网络相比更加灵活，对网络局部结构的更改不会影响到其余网络的连通性和性能。

## 致谢

经过两个多月的努力，我最终完成了毕业设计的编码和毕业论文的撰写工作。在此，我要感谢我的毕业设计和论文的指导老师，山东大学计算机与科学技术学院系统结构与高能计算研究所的朱方金老师。在代码编写的过程中，朱老师对我的每一个疑惑和难点又做了详细的解答，并在后续的论文编写过程中提出了指导性意见。在这里，请允许我表达对朱老师最真挚的感激。同时我还要感谢我身边优秀的同学们，没有你们的帮助和鼓励，我不可能完成这篇论文。

从一开始接触到毕设的题目时的一头雾水，到系统功能的实现，再到论文的完成。在这个过程中，我学到了很多知识和技能。我体会到了计算机网络的神奇之处，也感受到自己目前掌握的知识实在是凤毛麟角。这将激励我在以后的生活、学习和工作中，脚踏实地，不能眼高手低，树立终身学习的观念，不断通过实践来完善自己！

## 参考文献

- [1]<https://baike.baidu.com/item/ONF/823477>, 开放网络基金会
- [2]黄韬、刘江、魏亮主编,《软件定义网络核心原理与应用实践(第二版)》,人民邮电出版社,2016.09
- [3]<https://www.vmware.com/cn/support/support-resources/pubs.html>, VMware 官网文档.
- [4]<http://www.cqvip.com/QK/90580X/201308/46994096.html>, SDN 体系结构与未来网络体系结构创新环境
- [5]唐宏、刘汉江等主编,《OpenDaylight 应用指南》,人民邮电出版社,2016.01
- [6]雷葆华、王峰、王茜、王和宇主编,《SDN 核心技术剖析和实战指南》,电子工业出版社,2013.09
- [7]谢希仁主编,《计算机网络(第6版)》,电子工业出版社,2012.06
- [8]Open vSwitch 官网: <http://www.openvswitch.org/>
- [9]<http://mininet.org/download/>, Mininet 官网
- [10] <http://mininet.org/api/annotated.html>, Mininet Python API Reference Manual
- [11]<https://sflow.org/>, sFlow 官网
- [12]<https://www.python.org/>, Python 官网
- [13]<https://baike.baidu.com/item/NETCONF/4045177>, NETCONF
- [14][https://yeasy.gitbooks.io/mininet\\_book/runtime\\_and\\_example/mn.html](https://yeasy.gitbooks.io/mininet_book/runtime_and_example/mn.html), Mininet 源码解读
- [15]《Cloud based SDN and NFV architectures for IoT infrastructure》, by Mamdouh • Alenezi、Khaled • Almustafa and Khalim • AmjadMeerja.



## 附录 1 英文原文

Software defined network (SDN) based IoT architecture<sup>[15]</sup>。

In the new paradigm of software defined network (SDN) based virtualization, all the IoT network elements are simply forwarding devices without any intelligence instilled in them which can control and forward data traffic. These are simply COTS based equipment that receive commands from a separate software agent residing on remote servers. The entire network management and control operations reside in this software which is generally called SDN controller. The SDN controller is regarded as the brain of the entire network. The SDN controller resides on multiple physically distributed servers in a large cloud network. Besides residing on multiple servers, the SDN controller software behaves to logically control the network in a centralized manner. The control and management policies are seemed to be applied at the central location that reflects on the entire span of the network. This logical central control of the network will tremendously reduce the burden of network operators as it will avoid configuration errors across the network which is quite common in today's networks. Open and standard interfaces are developed between the data, control, and management planes that allows heterogeneous devices to connect to network without any effort. This is not possible with the current traditional networks where it is difficult to connect heterogeneous devices.

The three different planes namely data, control, and management planes in the SDN architecture are shown in Fig. 5. The data plane resides on the actual network hardware which are various COTS based IoT devices. The data plane is connected to the control plane through a southbound interface. The actual device virtualization takes place in the control plane residing in the SDN controller. Fig. 5 shows that the control plane in the SDN controller consists of a network hypervisor module for

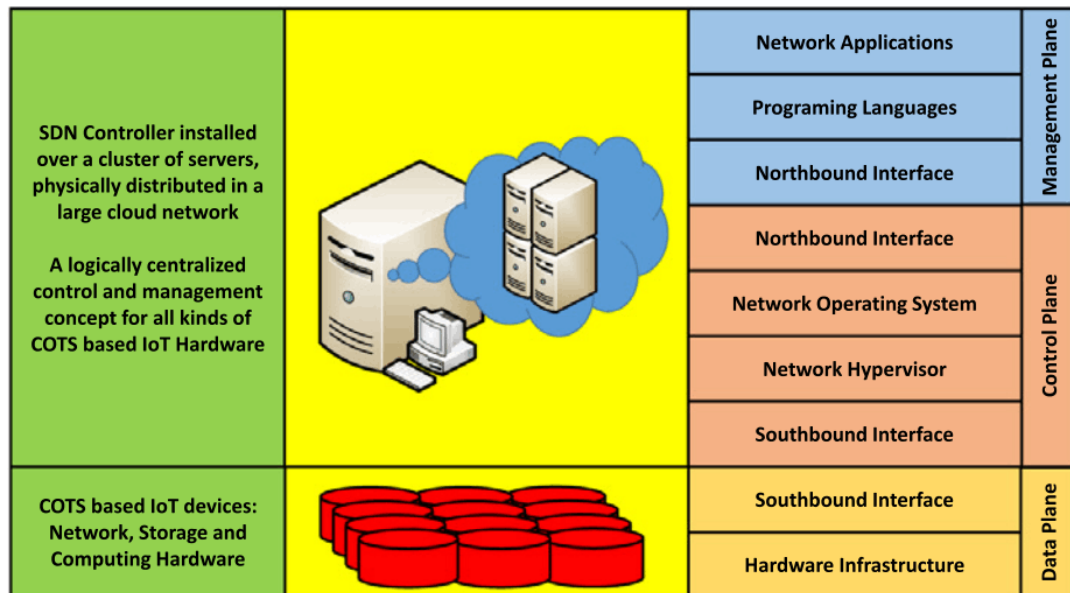
virtualization of the COTS based IoT devices. The SDN controller consists of both control and management planes as separate layers. These control and management planes communicate with each other using the northbound interface. The control plane also consists of the network operating system that controls the entire network as a single logical entity.

Network function virtualization (NFV) based IoT architecture.

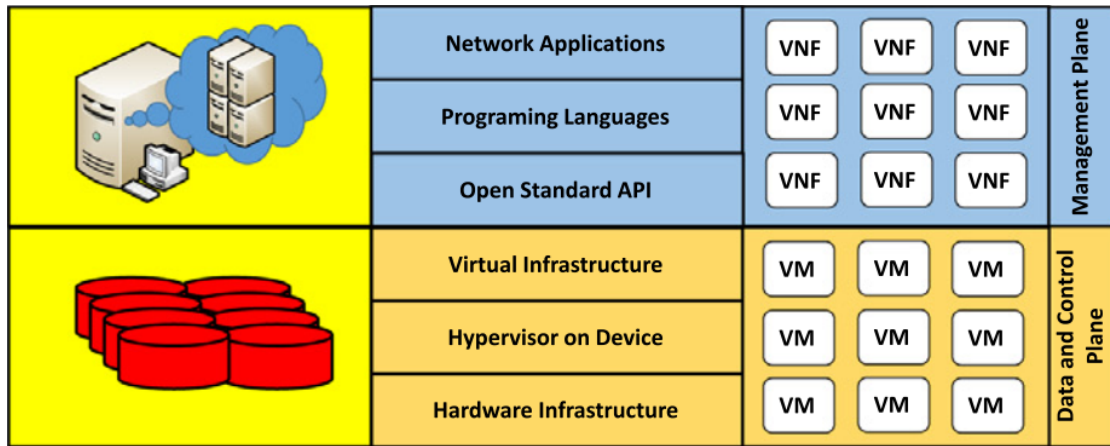
The conceptual diagram for network function virtualization (VNF) based architecture is shown in Fig. 5. Instead of a network hypervisor, this virtual layer in form of hypervisor is located on the device itself. The hypervisor creates virtual machines (VMs) on these physical hardware which is referred as virtual infrastructure in the conceptual diagram. The virtual hardware can be accessed using an open standard API. The higher level programming languages can access these standard set of open APIs to create virtual network functions (VNFs). The VNFs can be created using a central software manager running on a separate server farm. The resources can be allocated and released on the fly using a software manager similar to software controller in SDN architecture . On the otherhand, VNF enable devices can also be controlled using a central SDN controller so that both architectures can coexist and function together . The three basic components of the VNF architecture are: **(a) Physical Hardware:** The hardware is any bare-metal machine that hosts resources such as CPU, Memory, and storage. **(b) Virtual Hypervisor Layer:** This virtual layer is the software layer that runs on the bare-metal hardware that manages the resources such as CPU power, memory, and storage capacity. **(c) Virtual Machine:** The guest virtual machine is a software that emulates the architecture and functionalities of the physical platform using a fraction of hardware resources.

As a result, a particular physical hardware can host more than one VM. The maximum number of VMs that can be hosted on a physical hardware

is dependant on the resources of the physical hardware and the amount of resources used by each VM . The key advantage of VNF and SDN architectures is that a general purpose COTS based servers can be incorporated in enterprize class networks for Big Data handling and computation. Even the physical layer processing of the cellular mobile networks can be implemented in these COTS servers. This is a big step for telecommunication industry as it will transform the entire cellular network architecture. It will dramatically reduce the capital investment and reduce the energy consumption by resorting to cloud based data centers. However, it is yet to test the performance of such a network and only the future trials can be able to answer these questions through developing good test bed networks for active user trials. Multiple tenants will be able to share cloud based SDN and NFV architecture based virtualized network resources to improve profit margins and achieve reduced spending on infrastructure .



(a)SDN Architecture Kreutz et al.



(a) NFV Architecture.

## 附录 2 译文

基于软件定义网络（SDN）的物联网架构。

在基于软件定义网络（SDN）的新型虚拟化范例中，所有物联网网络元素都只是转发设备，而没有任何智能灌输其中，这些设备可以控制和转发数据流量。这些都是简单的基于 COTS 的设备，可以从驻留在远程服务器上的独立软件代理接收命令。整个网络管理和控制操作都驻留在这个通常称为 SDN 控制器的软件中。SDN 控制器被认为是整个网络的大脑。SDN 控制器驻留在大型云网络中的多个物理分布式服务器上。除了驻留在多台服务器上，SDN 控制器软件还可以集中管理网络。控制和管理政策似乎适用于反映整个网络范围的中心位置。网络的这种逻辑中央控制将极大地减轻网络运营商的负担，因为它将避免当今网络中常见的网络配置错误。在数据，控制和管理平面之间开发开放和标准接口，允许异构设备无需任何努力即可连接到网络。这对于当前难以连接异构设备的传统网络是不可能的。

SDN 架构中的三个不同的平面，即数据，控制平面和管理平面如图 5 所示。数据平面驻留在实际的网络硬件上，这些硬件是各种基于 COTS 的 IoT 设备。数据平面通过南向接口连接到控制平面。实际的设备虚拟化发生在驻留在 SDN 控制器中的控制平面中。图(a)显示 SDN 控制器中的控制平面由用于虚拟化基于 COTS 的 IoT 设备的网络管理程序模块组成。SDN 控制器由控制和管理平面组成，作为单独的层。这些控制和管理平面使用北向接口相互通信。控制平面还包

括将整个网络控制为单个逻辑实体的网络操作系统。

基于网络功能虚拟化（NFV）的物联网架构。

基于网络功能虚拟化（VNF）架构的概念图如图(b)所示。虚拟管理程序形式的虚拟层位于设备本身上，而不是网络管理程序。管理程序在这些物理硬件上创建虚拟机（VM），在概念图中将其称为虚拟基础架构。虚拟硬件可以使用开放标准 API 进行访问。更高级别的编程语言可以访问这些标准的开放 API 集合来创建虚拟网络功能（VNF）。可以使用在单独的服务器场上运行的中央软件管理器来创建 VNF。资源可以使用类似于 SDN 架构中的软件控制器的软件管理器实时分配和发布。另一方面，也可以使用中央 SDN 控制器来控制 VNF 启用设备，以便两种体系结构可以共存并共同运行。VNF 架构的三个基本组件是：

（a）物理硬件：硬件是承载 CPU，内存和存储等资源的任何裸机。

（b）虚拟管理程序层：该虚拟层是运行在裸机硬件上的软件层，用于管理诸如 CPU 功率，内存和存储容量等资源。

（c）虚拟机：来宾虚拟机是一种使用少量硬件资源来模拟物理平台的体系结构和功能的软件。

因此，特定的物理硬件可以托管多个 VM。可以在物理硬件上托管的最大 VM 数量取决于物理硬件资源和每个 VM 使用的资源数量。VNF 和 SDN 体系结构的关键优势在于，基于 COTS 的通用服务器可以集成到企业级网络中，用于大数据处理和计算。甚至蜂窝移动网络的物理层处理也可以在这些 COTS 服务器中实现。这对电信行业来说是一个巨大的进步，因为它将改变整个蜂窝网络架构。它将通过采用基于云的数据中心来大幅降低资本投资并降低能耗。然而，它尚未测试这种网络的性能，只有未来的试验才能通过开发用于积极用户试验的良好测试床网络来回答这些问题。多租户将能够共享基于云的 SDN 和基于 NFV 架构的虚拟化网络资源，以提高利润率并减少基础设施开支。