OpenDaylight 指南

最新版: yeasy@github

更新历史:
V0.5: 2013-12-30
    增加对 VTN 项目的分析；
    部分格式调整。
V0.4: 2013-08-22
    添加如何在控制台调用 bundle 方法。
V0.3: 2013-08-20
    添加增量项目编译说明；
    优化结构。
V0.2: 2013-07-14
    完成用户安装运行和开发环境部署。
V0.1: 2013-07-10
    完成框架。

第1章 概述

OpenDaylight项目目前包括6个Bootstrap项目：OpenDaylight Controller、OpenDaylight Network Virtualization Platform、OpenDaylight Virtual Tenant Network (VTN)、Open DOVE OpenFlow Plugin、Affinity Metadata Service，以及9个Incubation项目：YANG Tools、LISP Flow Mapping、OVSDB Integration、OpenFlow Protocol Library、BGP-LS/PCEP、Defense4All、SNMP4SDN、dlux - openDayLight User eXperience、SDN Simulation Platform。

第2章 控制器安装运行

官方网站为 http://www.Open Daylight.org。

2.1 环境配置

Open Daylight Controller 是 java 程序，理论上可以运行在任何支持 java 的环境中。推荐环境为比较新的 Linux 环境和 Java 虚拟机 1.7+版本。

例如，在 Ubuntu 12.04 系统中，需要安装 java1.7 版本，命令为：

```
#sudo add-apt-repository ppa:webupd8team/java
#sudo apt-get update && sudo apt-get install oracle-jdk7-installer
```

之后，配置 JAVA_HOME 环境变量，添加下面的行到/etc/environment 文件。

```
JAVA_HOME=/usr/lib/jvm/java-7-oracle
```

2.2 安装和使用

2.2.1 获取源码

源码可以从 https://jenkins.Open Daylight.org/controller/job/controller-nightly/lastSuccessfulBuild/artifact/Open Daylight/distribution/Open Daylight/target/下载，或者利用 git 下载最新的版本

对于匿名用户：

```
git clone https://git.Open Daylight.org/gerrit/p/controller.git
```

对于项目注册用户：

```
git clone ssh://<username>@git.Open Daylight.org:29418/controller.git
```

2.2.2 编译运行

进入 Open Daylight/distribution/Open Daylight/src/main/resources 目录，下面一般有若干文件和目录，包括：

run.bat：Windows 系统下的运行脚本。

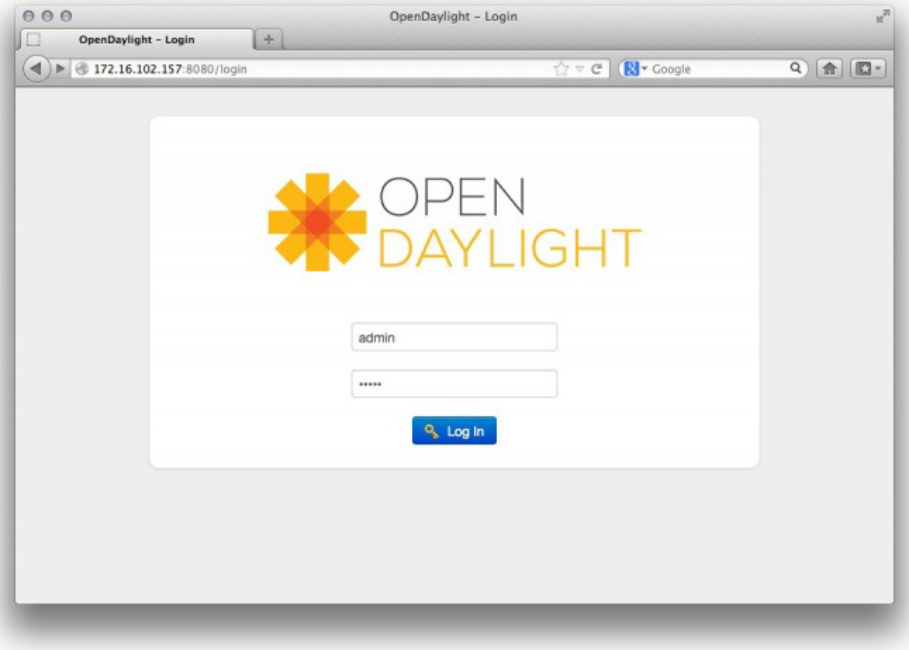
run.sh：Linux 系统下的运行脚本。

version.properties：编译的版本信息。

configuration：基本的配置信息。

在 Linux 平台上，需要用 root 权限运行./run.sh。

运行成功后可以打开浏览器访问本地地址，并登陆控制器 web UI，默认用户名和密码都是 admin。



图表 1 控制器 Web 登陆界面

2.2.3 常见运行问题

如果运行时报错：java.lang.OutOfMemoryError: PermGen space。可以通过修改 maven 配置为

```
export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
```

2.2.4 使用 Mininet 测试

Mininet 可以用来模拟大规模的虚拟网络。可以从 mininet.github.org 下载代码。

Open Daylight Controller 可以跟 Mininet 很好的联动。

在已经安装 Mininet 环境的机器（如果跟 controller 在同一个机器，可以用 127.0.0.1）中，启动 Mininet

```
mininet@mininet-vm:~$ sudo mn --controller=remote，ip=127.0.0.1 --topo tre
e，3
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7
*** Adding links:
```

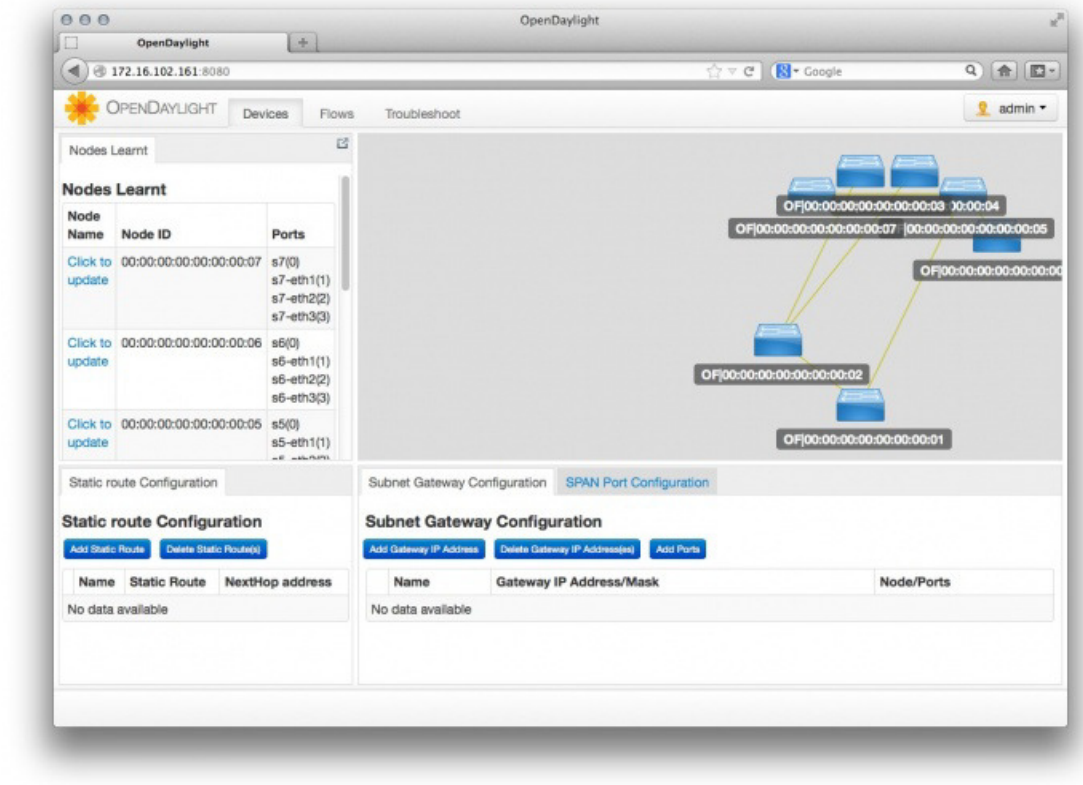
```
(h1， s3) (h2， s3) (h3， s4) (h4， s4) (h5， s6) (h6， s6) (h7， s7) (h8， s7) (s1
， s2) (s1， s5) (s2， s3) (s2， s4) (s5， s6) (s5， s7)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7
*** Starting CLI:
mininet>
```

连接成功后可以测试控制器的功能。

2.2.4.1 Simple Forwarding Application

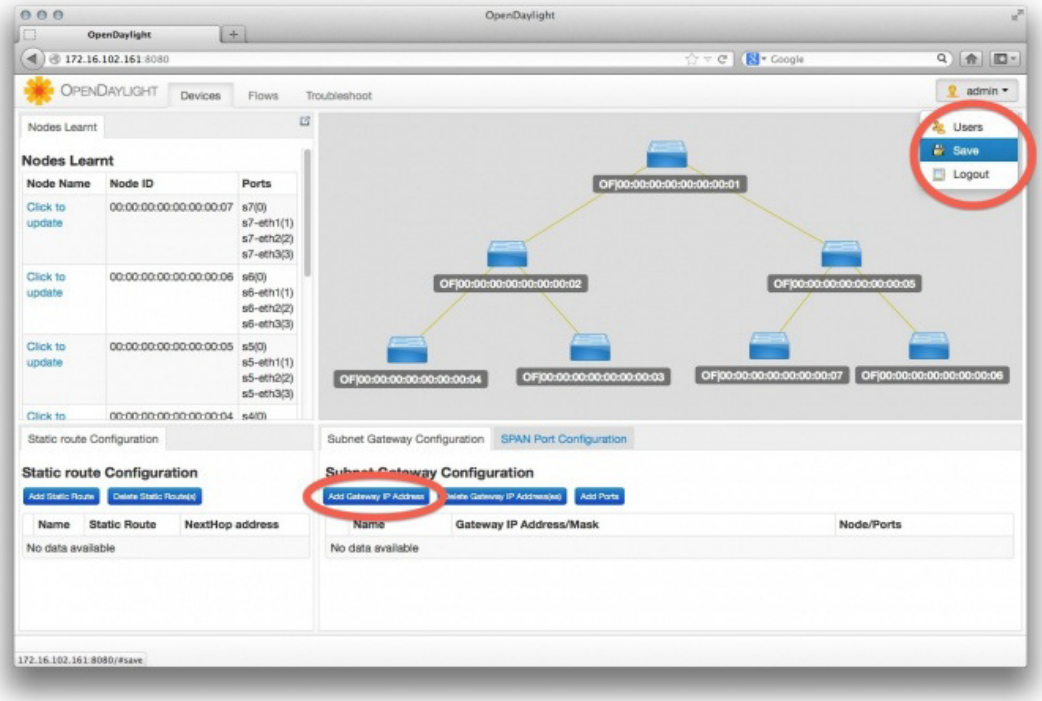
Simple Forwarding 是 Open Daylight Controller 上的一个应用，它通过 arp 包来探测连接到网络的每个主机，并给交换机安装规则（该规则指定到某个主机地址的网包路径），让网包能顺利转到各个主机。

首先，登录到 Controller 界面。



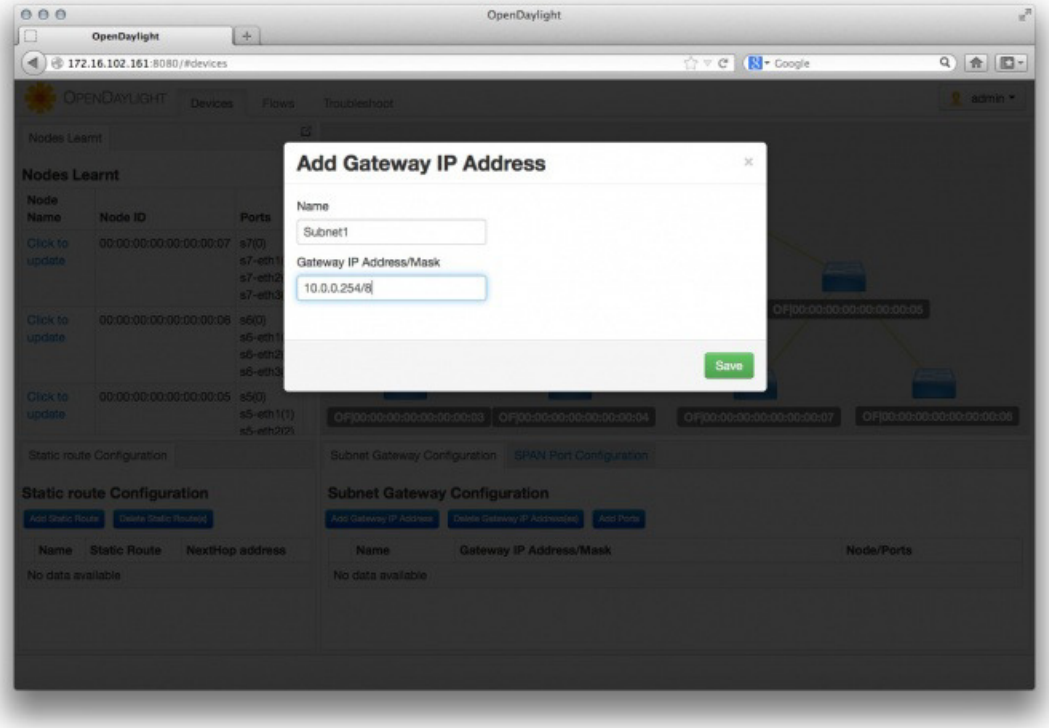
图表 2 Simple Forwarding 界面

通过拖动设备，形成逻辑拓扑，并保存配置。



图表 3 Simple Forwarding 拖动成逻辑拓扑

点击 Add Gateway IP Address 按钮，添加 IP 和 10.0.0.254/8 子网。

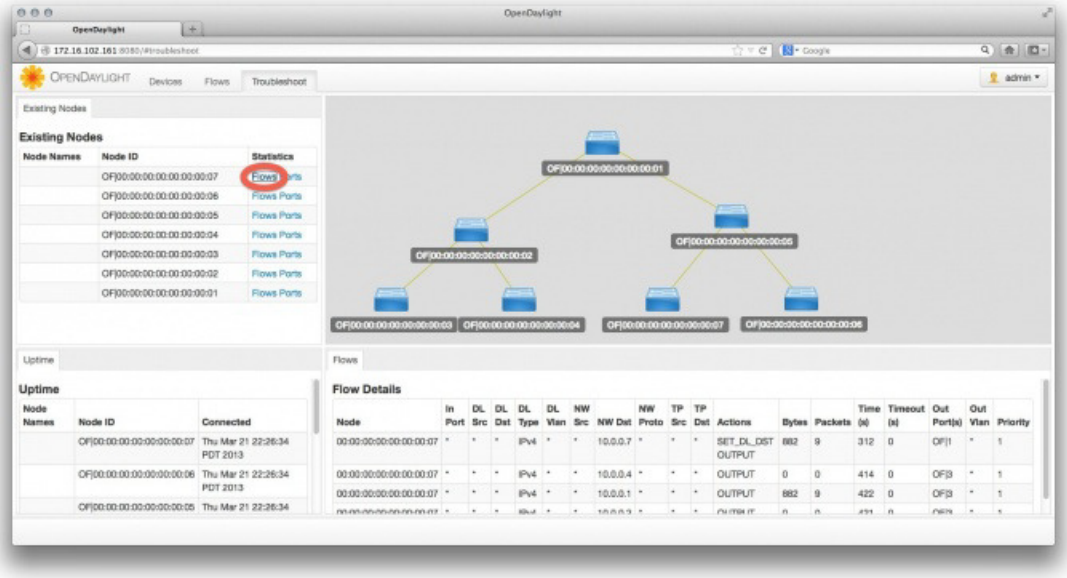


图表 4 添加网关 IP

在 Mininet 中确认主机可以相互连通。

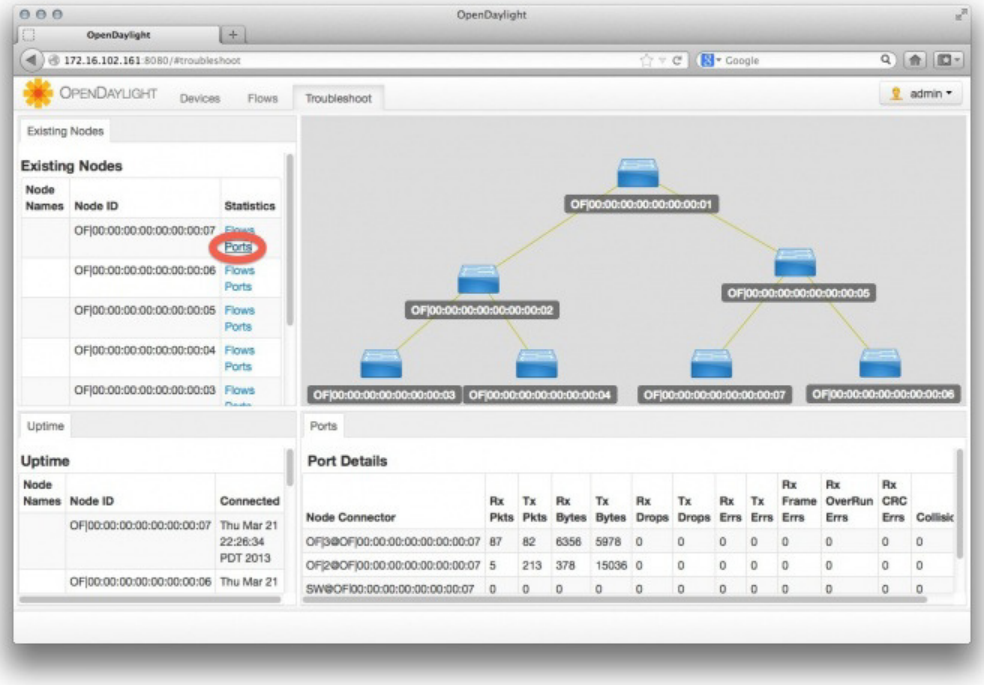
```
mininet> h1 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_req=1 ttl=64 time=1.52 ms
64 bytes from 10.0.0.7: icmp_req=2 ttl=64 time=0.054 ms
64 bytes from 10.0.0.7: icmp_req=3 ttl=64 time=0.060 ms
64 bytes from 10.0.0.7: icmp_req=4 ttl=64 time=0.052 ms
--- 10.0.0.7 ping statistics ---
4 packets transmitted， 4 received， 0% packet loss， time 2999ms
rtt min/avg/max/mdev = 0.052/0.422/1.523/0.635 ms
mininet>
```

点击 Troubleshooting 标签页，查看交换机的流表细节。



图表 5 查看交换机流表细节

查看端口细节。



图表 6 查看端口细节

在 osgi 的控制台，输入 ss simple，可以看到 Simple Forwarding 应用被激活。

```
osgi> ss simple
"Framework is launched."


id    State       Bundle
45    ACTIVE      org.Open Daylight.controller.samples.simpleforwarding_0.4.0.S
NAPSHOT
```

第3章 开发环境

3.1 概述

Open Daylight Controller 提供了一个模块化的开放 SDN 控制器。

它提供了开放的北向 API（开放给应用的接口），同时南向支持包括 OpenFlow 在内的多种 SDN 协议。底层支持混合模式的交换机和经典的 OpenFlow 交换机。

运行系统推荐为 Linux 系列，并安装了 Java1.7+环境。

自带的 Web UI 也是利用了北向的 API 进行交互。

3.2 架构原则

Open Daylight Controller 在设计的时候遵循了六个基本的架构原则：

运行时模块化和扩展化（Runtime Modularity and Extensibility）：支持在控制器运行时进行安装、删除和服务的更新。

多协议的南向支持（Multiprotocol Southbound）：南向支持多种协议。

服务抽象层（Service Abstraction Layer）：南向多种协议对上提供统一的北向服务接口。

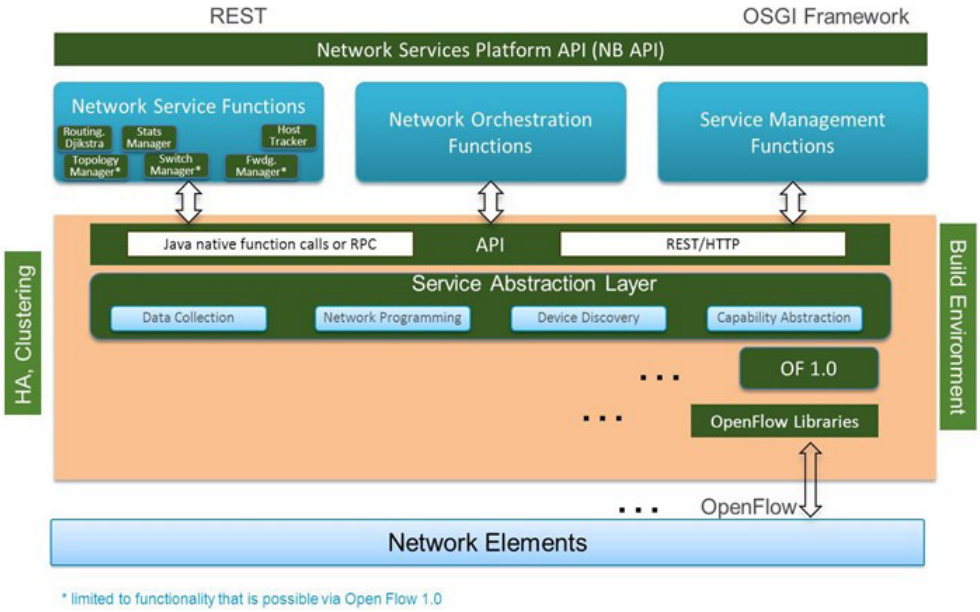
开放的可扩展北向 API（Open Extensible Northbound API）：提供可扩展的应用 API，通过 REST 或者函数调用方式。两者提供的功能要一致。

支持多租户、切片（Support for Multitenancy/Slicing）：允许网络在逻辑上（或物理上）划分成不同的切片或租户。控制器的部分功能和模块可以管理指定切片。控制器根据所管理的分片来呈现不同的控制观测面。

一致性聚合（Consistent Clustering）：提供细粒度复制的聚合和确保网络一致性的横向扩展（scale-out）。

3.3 架构框架

3.3.1 框架概述



图表 7 架构框架

如图表 7 所示，南向通过 plugin 的方式来支持多种协议，包括 OpenFlow1.0、1.3，BGP-LS 等。这些模块被动态挂载到服务抽象层（SAL），SAL 为上层提供服务，将来自上层的调用封装为适合底层网络设备的协议格式。控制器需要获取底层设备功能、可达性等方面的信息，这些信息被存放在拓扑管理器（Topology Manager）中。其他的组件，包括 ARP handler、Host Tracker、Device Manager 和 Switch Manager，则为 Topology Manager 生成拓扑数据。

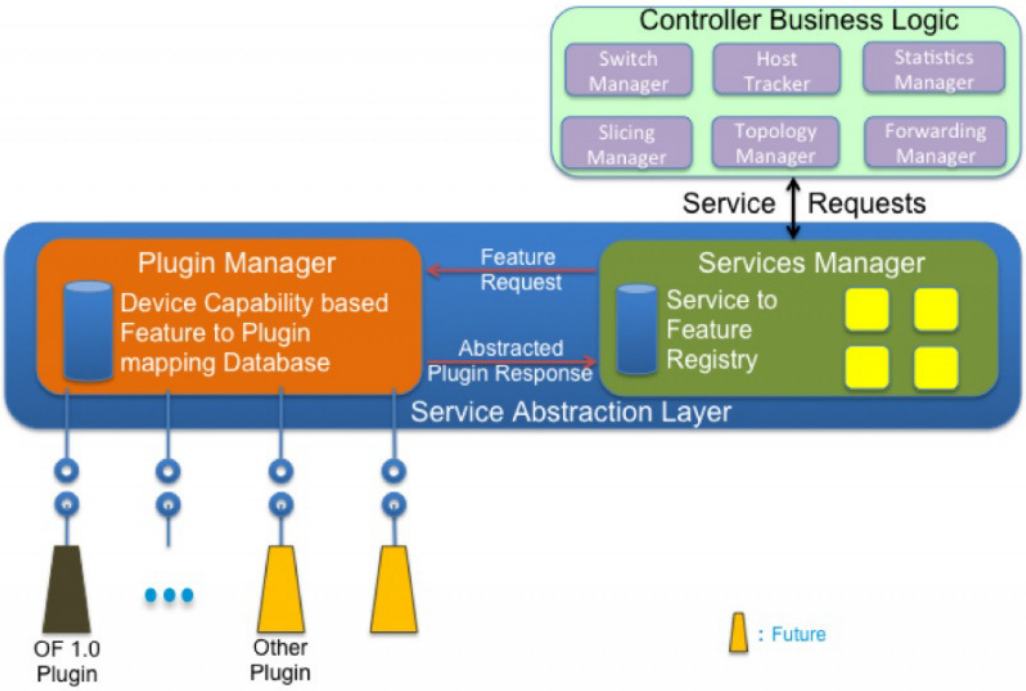
控制器为应用（App）提供开放的北向 API。支持 OSGi 框架和双向的 REST 接口。OSGi 框架提供给与控制器运行在同一地址空间的应用，而 REST API 则提供给运行在不同地址空间的应用。所有的逻辑和算法都运行在应用中。

控制自带了 GUI，这个 GUI 使用了跟应用同样的北向 API，这些北向 API 也可以被其他的应用调用。

3.3.2 功能概述

3.3.2.1 服务抽象层

服务抽象层（SAL）是整个控制器模块化设计的核心，支持多种南向协议，并为模块和应用支持一致性的服务。



图表 8 抽象服务层

OSGi 框架支持动态链接插件，来支持更多的南向协议。SAL 提供了基本的服务，比如设备探测（用于拓扑管理器）。服务基于插件提供的特性来构建。服务的请求被 SAL 映射到合适的插件上，采用合适的南向协议跟底层设备进行交互。各个插件相互之间独立并且跟 SAL 松耦合。SAL 框架是 Open Daylight Controller 项目的贡献之一。

拓扑服务（Topology service）是一系列服务，允许传输拓扑信息，包括探测新加入的节点和链路等；

数据包服务（Data Packet service）将来自代理的数据包发送给应用；

流编程服务（Flow Programming service）为不同的代理编程流规则提供必要的逻辑。

统计服务（Statistics service）搜集统计信息。

流

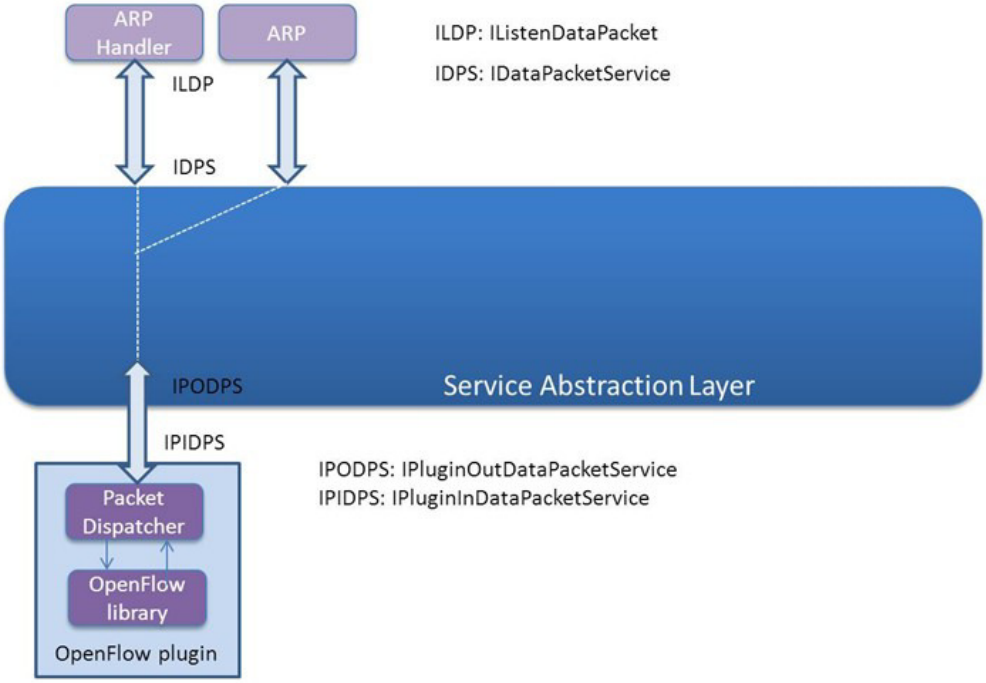
节点连接（Node Connector）或者端口。

队列。

清单服务（Inventory service），返回节点和端口的存储信息。

资源服务（Resource service）查询资源的状态。

3.3.2.2 SAL 服务：数据包服务



图表 9 数据包服务

图表 9 展示了一个实现 SAL 服务的例子，基于 OpenFlow1.0 来实现数据包服务。

- IListenDataPacket 是一个被上层模块或应用（例如 ARP Handler）实现的服务，用于接收数据包。
- IDataPacketService 接口被 SAL 实现，提供从代理发送和接收数据包的服务。这个服务在 OSGi 中注册，以便其他程序获取调用。
- IPluginOutDataPacketService 接口面向 SAL，用于当一个协议插件希望发送一个网包到应用层。
- IPluginInDataPacketService 接口面向协议插件，用于通过 SAL 向代理发送网包。

代码在 SAL 和各个服务、插件之间的执行过程：

OpenFlow 插件获取了 ARP 包，该包需要被 ARP Hanlder 应用进行处理；

OpenFlow 插件调用 IPluginOutDataPacketService 将包发送到 SAL；

ARP Handler 应用实现已经注册到了 IListenDataPacket 服务，SAL 收到网包后将把包发送到 ARP Handler 应用；

ARP Handler 应用将处理网包。

反向的处理过程（发送出去网包）是

应用创建包并调用 SAL 提供的 IDataPacketService 接口发出包。目标网络设备作为 API 参数的一部分；

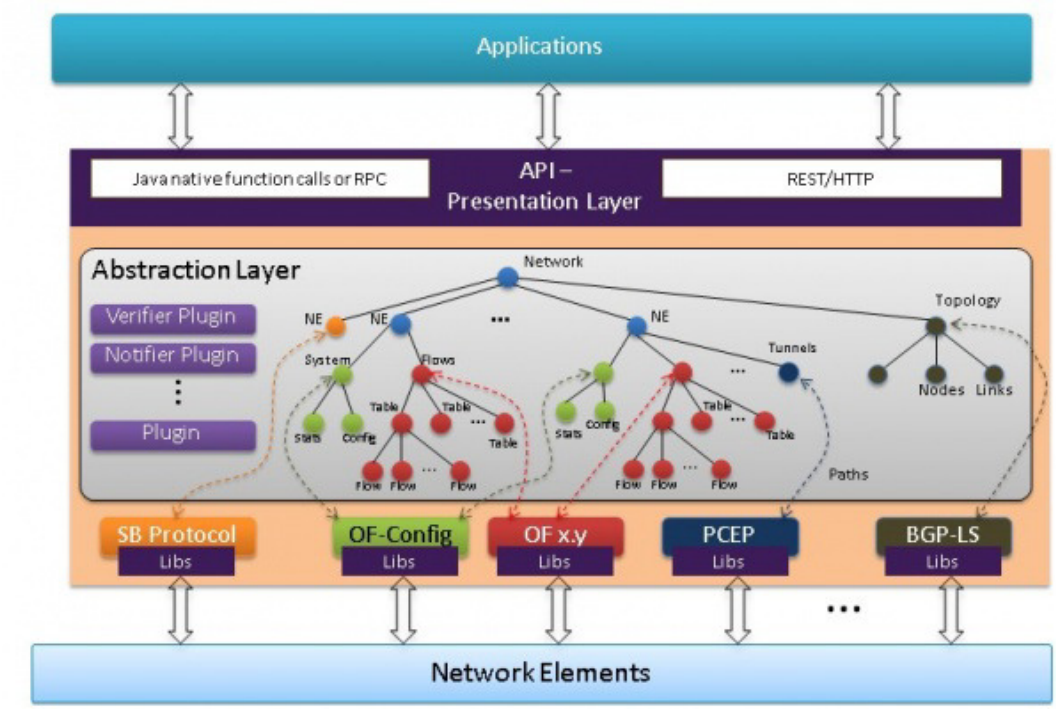
SAL 根据目标网络设备为对应的协议插件（例如 OpenFlow 插件）调用 IPluginInDataPacketService 接口。

协议插件将包发送给适当的网络元素。协议相关的处理都由插件完成。

3.3.2.3 控制器 SAL 的演进

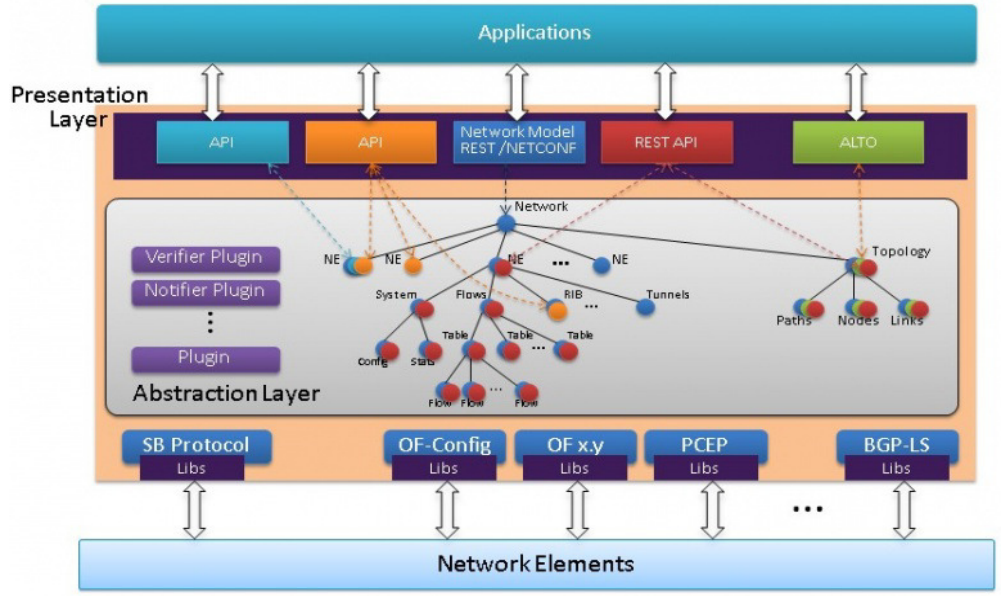
SAL 已经演进成为了一种基于模型的方法。框架被用于对网络（属性和设备）建模，并动态地在服务应用之间通过北向 API 和协议插件提供的南向 API 进行映射。图表 10 展示了南向

插件如何提供网络模型树的部分。



图表 10 南向插件提供网络模型树的部分

图表 11 展示了应用如何通过北向 API 来获取网络模型的信息。



图表 11 应用通过北向 API 来获取网络模型信息

3.3.2.4 交换机管理器

Switch Manager API 处理网络元素的细节。当网络元素被探测到的时候，它的属性以 Switch Manager 的形式存放在数据库中。

3.3.2.5 GUI

GUI 作为一个应用来实现，采用了北向的 REST API 来跟其他模块交互。因此，GUI 能实现的功能很容易被集成到其他的管理系统中。

3.3.2.6 高可靠性

Open Daylight 控制器支持一个基于集群的高可用性模型。多个 Open Daylight 控制器可以配合作为一个逻辑控制器。不仅支持了细粒度的冗余，并且支持线性的横向扩展。为了支持高可用性，需要在下面几个方面增加回弹性能：

- 控制器层：通过集群方式添加一个或多个控制器实例。
- 交换机支持 multi-homed，支持多控制器。
- 应用支持 multi-homed，支持多控制器。

交换机通过持续的点到点 TCP/IP 协议连接到多个控制器。控制器和应用之间通过 REST 接口连接，是基于 HTTP 的。因此所有的基于 HTTP 的回弹特性将可以被利用，包括：

- 为控制器集群提供一个虚拟 IP。
- 利用轮询 DNS 查询后，应用跟集群之间发送交互。
- 在应用和控制器集群之间部署一个 HTTP 的负载均衡，不仅提供可靠性，还可以根据请求 URL 来提供负载均衡。

在 OpenFlow1.2 规范中定义了交换机对 multi-home 的支持，包括两种操作模型：

- 对等交互：所有的控制器都能读写交换机，需要之间进行同步；
- 主从交互：存在一个主控制器和多个从控制器。

以上两种模型都可以采用。在主从模型情况下，会容易避免相互竞争造成的逻辑紊乱。在 OpenFlow1.0 中不支持 multi-home，因此可以作为扩展来实现。

对于多个控制器之间的相互交互，需要同步以下的信息：

- 内存数据库中的拓扑信息；
- 交换机和主机在数据库中的记录；
- 配置文件；
- 交换机的主控制器；
- 用户数据库。

一般假设在各个节点上的路径计算是各自独立的。如果需要保持一致性，那么路径信息也需要进行同步。

使用 REST API 的应用在应用和控制器之间采用了非持久性连接，因此，当控制器断线后，应用将重新发起新的连接。如果在传输中发生失败，则将产生 HTTP 错误并采取纠错行为。

对于使用 OSGi 框架的应用来说，应用在某个控制器上运行，如果控制器断线，那么应用也随之停止工作。应用需要自己负责在多个控制器存在时的可靠性和多实例之间的状态同步。

控制器提供集群服务，多个控制器之间可以同步状态和事件。同时提供了交互（transaction）API 来维护集群中节点之间的交互。

3.4 开发框架概述

包括如下几个部分。

版本管理：git：https://git.opendaylight.org/

代码 review：Gerrit：https://git.opendaylight.org/gerrit/

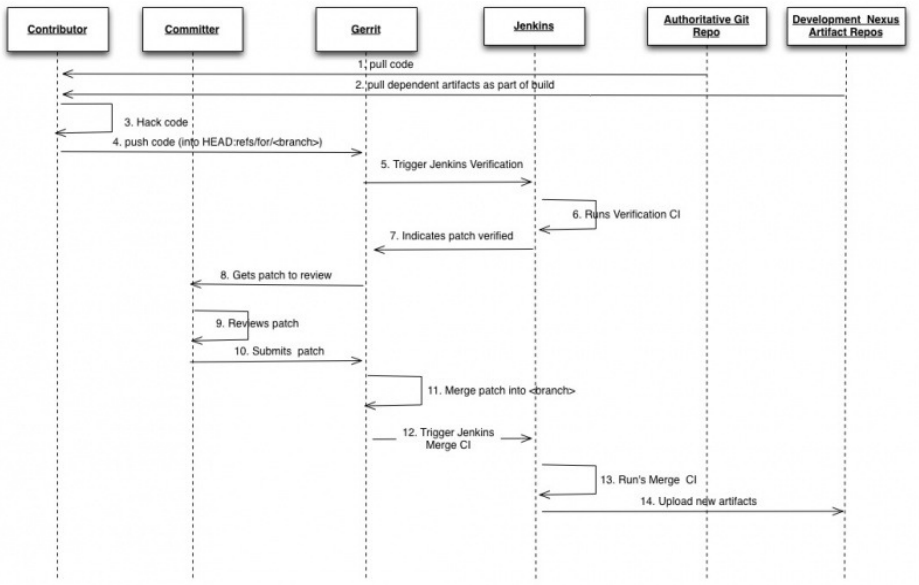
连续集成：Jenkins：https://jenkins.opendaylight.org/

人工 Repo：Nexus：https://nexus.opendaylight.org/

质量管理：Sonar：https://nexus.opendaylight.org/

Bug 跟踪：Bugzilla：http://bugs.opendaylight.org/

Wiki：MediaWiki：https://wiki.opendaylight.org/
整体的代码流如图表 12 所示。



图表 12 整体代码流

首先，用户通过认证的 git 方式来获取代码；
从开发 Nexus 的库中获取需要的依赖；
修改和编写代码；
提交代码；
触发 Jenkins 审查；
进行连续集成；
提交 patch；
审查 path；
提交 patch；
合并 patch 到 branch 中；
触发 Jenkins 的合并 CI；
检查需要的新的依赖；
提交新的依赖。

3.5 开发和更新代码

3.5.1 通过 CLI

用 git 从库中获取代码：

```
git clone ssh://<username>@git.opendaylight.org:29418/controller.git
```

匿名获取代码可以执行：

```
git clone https://git.opendaylight.org/gerrit/p/controller.git
```

配置 Gerrit Change-id 提交信息。

```
cd controller
scp -p -P 29418 <username>@git.opendaylight.org:hooks/commit-msg .git/hooks/
```

```
chmod 755 .git/hooks/commit-msg
```

编译代码

```
cd opendaylight/distribution/opendaylight/
mvn clean install [-DskipTests]
```

运行控制器

```
cd  target/distribution.opendaylight-0.1.0-SNAPSHOT-osgipackage/opendaylight/
./run.sh
```

提交代码

```
git commit --signoff
```

从代码库获取最新代码：

```
git pull ssh://<username>@git.opendaylight.org:29418/controller.git HEAD:refs/for/master
```

推送代码到代码库：

```
git push  ssh://<username>@git.opendaylight.org:29418/controller.git HEAD:refs/for/master
```

之后可以登录 Gerrit 查看提交信息。

注意：编译整个项目可能需要比较长的时间，如果不需要进行测试，并且不需要清理原先的编译结果，可以执行

```
mvn  install –DskipTests
```

更进一步的，如果只是想更新某个 bundle，可以不用编译和停止 OSGi 框架，直接进入 bunlde 子目录，执行 mvn 编译，编译成功后将 jar 文件复制到整体项目的 plugin 目录下，并注意重命名。此时在 OSGi 框架中可以通过 start 和 stop 命令来重启该 bundle，应用更新。

例如

```
cd ~/controller/opendaylight/arphandler
mvn clean install
cp target/arphandler-0.4.0-SNAPSHOT.jar ../distribution/opendaylight/target/distribution.opendaylight-0.1.0-SNAPSHOT-osgipackage/opendaylight/plugins/org.opendaylight.controller.arphandler-0.4.0-SNAPSHOT.jar
```

3.5.2 通过 Eclipse

3.5.2.1 配置 eclipse

首先通过 git 下载代码。

打开 eclipse，安装 maven 插件。

Eclipse->help->install new software。

添加源 http://download.eclipse.org/technology/m2e/releases，搜索 m2e 和 m2e-slf4j，都安装 1.2.0 版本。然后完成安装。

重启 eclipse。

导入 git 下载的项目。

Eclipse->file->import->maven-> Existing Maven Projects，找到 opendaylight/distribution/opendaylight 目录，完成。如果询问是否安装 Tycho，选择是。

3.5.3 在 OSGi 控制台中调用方法

模块可以通过实现 CommandProvider 接口来注册自身的方法，让用户通过 OSGi 的控制台命令直接调用。步骤如下（请参考 Switchmanager.implementation）：

在 bundle 的 POM.xml 文件中的 Import-Package 列表中，加入 org.eclipse.osgi.framework.console 和 org.osgi.framework 包。

在实现类中，继承 CommandProvider 接口。

重载 getHelp()函数，这样在 OSGi 控制台输入 help 命令的时候会输出相关的提示。

此时，所有类内的 public void _<command_name>(CommandInterpreter ci)格式的函数将被 OSGi 控制台中的<command_name>触发。

实现注册方法，并在 start()方法中进行调用。

例如，注册方法为

```
private void registerWithOSGIConsole() {
        BundleContext bundleContext =
FrameworkUtil.getBundle(this.getClass()).getBundleContext();
        bundleContext.registerService(CommandProvider.class.getName(),  this,
null);
  }
```

3.6 示例应用

3.6.1 Simple Forwarding Application

3.6.2 Statistics Application

3.6.2.1 JAXB 统计客户端

下面介绍一个基于 JAXB 的应用，使用了统计模块。

```
package org.opendaylight.controller.topology;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;

import org.apache.commons.codec.binary.Base64;
import org.opendaylight.controller.sal.reader.FlowOnNode;
```

```
import org.opendaylight.controller.statistics.northbound.AllFlowStatistics;
import org.opendaylight.controller.statistics.northbound.FlowStatistics;

public class JAXBStatisticsClient {

 public static void main(String[] args) {

 System.out.println("Starting Statistics JAXB client.");

 String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
 String containerName = "default";
 String user = "admin";
 String password = "admin";

 URL url;
 try {
 url = new java.net.URL(baseURL + "/" + containerName + "/flowstats");

 String authString = user + ":" + password;
 byte[] authEncBytes = Base64.encodeBase64(authString.getBytes());
 String authStringEnc = new String(authEncBytes);
 URLConnection connection = url.openConnection();
 connection.setRequestProperty("Authorization", "Basic "
 + authStringEnc);

 connection.setRequestProperty("Content-Type", "application/xml");
 connection.setRequestProperty("Accept", "application/xml");

 connection.connect();

 JAXBContext context = JAXBContext.newInstance(AllFlowStatistics.class);
 Unmarshaller unmarshaller = context.createUnmarshaller();

 InputStream inputStream = connection.getInputStream();
```

```
AllFlowStatistics result = (AllFlowStatistics) unmarshaller.unmarshal(inputStrea
m);

System.out.println("We have these statistics:");

for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t" + flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
}

} catch (Exception e) {
System.out.println(e.getLocalizedMessage());
}
}
}
```

下面具体分析代码的工作过程，首先是创建连接，练到统计模块上。

```
String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
String containerName = "default";
String user = "admin";
String password = "admin";

URL url;
try {
url = new java.net.URL(baseURL + "/" + containerName + "/flowstats");

String authString = user + ":" + password;
byte[] authEncBytes = Base64.encodeBase64(authString.getBytes());
```

```
String authStringEnc = new String(authEncBytes);
URLConnection connection = url.openConnection();
connection.setRequestProperty("Authorization", "Basic "
+ authStringEnc);

connection.setRequestProperty("Content-Type", "application/xml");
connection.setRequestProperty("Accept", "application/xml");

connection.connect();
```

为了获取流统计信息，需要创建一个 JAXB 上下文，然后数据被传输到 AllFlowStatistics 对象。

```
for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t" + flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
}
```

3.6.2.2 Jersey 统计客户端

跟基于 JAXB 的统计客户端类似，代码为

```
package org.opendaylight.controller.topology;

import org.opendaylight.controller.sal.reader.FlowOnNode;
import org.opendaylight.controller.statistics.northbound.AllFlowStatistics;
import org.opendaylight.controller.statistics.northbound.FlowStatistics;

import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.filter.HTTPBasicAuthFilter;
```

```
public class JerseyStatisticsClient {

public static void main(String[] args) {

System.out.println("Starting Topology JAXB client.");

String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
String containerName = "default";
String user = "admin";
String password = "admin";

try {

Client client = com.sun.jersey.api.client.Client.create();
client.addFilter(new HTTPBasicAuthFilter(user， password));

AllFlowStatistics result = client.resource(
baseURL + "/" + containerName + "/flowstats").get(
AllFlowStatistics.class);

System.out.println("We have these statistics:");

for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t"
+ flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
}
```

```
} catch (Exception e) {
System.out.println(e.getLocalizedMessage());
}
}
}
```

下面分析代码过程，首先，也是需要连接到统计模块。

```
String baseURL = "http://127.0.0.1:8080/one/nb/v2/statistics";
String containerName = "default";
String user = "admin";
String password = "admin";

try {

Client client = com.sun.jersey.api.client.Client.create();
client.addFilter(new HTTPBasicAuthFilter(user， password));
```

为了获取流信息，需要创建一个 Jersey 客户端对象，数据被存放到 AllFlowStatistics 对象中。

```
AllFlowStatistics result = client.resource(
baseURL + "/" + containerName + "/flowstats").get(
AllFlowStatistics.class);
```

AllFlowStatistics 对象被转化为一个 FlowStatistics 对象，结果从 FlowStatistic 对象中获取，可以通过 get 方法来实现。

```
for (FlowStatistics statistics : result.getFlowStatistics()) {
System.out.println(statistics.getNode().getNodeIDString());
System.out.println(statistics.getNode().getType());
for (FlowOnNode flowOnNode : statistics.getFlowStat()) {
System.out.println("\t" + flowOnNode.getByteCount());
System.out.println("\t"
+ flowOnNode.getDurationNanoseconds());
System.out.println("\t" + flowOnNode.getDurationSeconds());
System.out.println("\t" + flowOnNode.getPacketCount());
System.out.println("\t" + flowOnNode.getTableId());
System.out.println("\t" + flowOnNode.getFlow());
}
```

```
}
```

3.6.3 Load Balancer Application

应用可以通过源地址和源端口将流量负载均衡到后端的服务器上。该服务被动的安装流表规则，将所有的特定来源地址和端口的数据包发送到合适的后端服务器上。该服务可以通过 REST API 来被外部程序配置。

要使用这个服务，首先所有客户端需要使用一个 VIPP 作为目标地址。VIP 包括虚拟 IP、端口和协议。

一些前提假设：

- 同一个服务器池可以被分配一个或多个 VIP，但同一个池必须使用同样的均衡策略。
- 对于每一个 VIP，最多分配一个服务器池。
- 所有的流表项默认超时时间为 5 秒。
- 到达 VIP 的网包在离开 OpenFlow 机器的时候必须从进入的交换机离开。
- 删除 VIP、服务器池，或从池中删除一个服务器的时候，服务并不删除之前添加的流表项，这些流表项通过自动超时来删除。

3.6.3.1 REST API

表格 1 负载均衡服务的 REST API

Description	URI	Type	Request Body/Arguments	Response Codes
List details of all existing pools	/one/nb/v2/lb/{container-name*}/	GET		200 ("Operation successful") 404 ("The containerName is not found") 503 "Load balancer service is unavailable")
List details of all existing VIPs	/one/nb/v2/lb/{container-name}/vips	GET		200 ("Operation successful") 404 ("The containerName is not found") 503 ("Load balancer service is unavailable")

Create pool	/one/nb/v2/lb/{container-name}/create/pool	POST	{
			"name":"",
			"lbmethod":""
			}
				201 ("Pool created successfully")
				404 ("The containerName not found")
				503 ("Load balancer service is unavailable")
				409 ("Pool already exist")
				415 ("Invalid input data")

Delete pool	/one/nb/v2/lb/{container-name}/delete/pool/{pool-name}	DELETE		200 ("Pool deleted successfully") 404 ("The containerName not found") 503 ("Load balancer ser

				vice is unavailable") 404 ("Pool not found") 500 ("Failed to delete pool")

Create VIP	/one/nb/v2/lb/{container-name}/create/vip	POST	{
			"name":"",
			"ip":"ip in (xxx.xxx.xxx.xxx) format",
			"protocol":"TCP/UDP",
			"port":"any valid port number",
			"poolname":"" (optional)
			}
				201 ("VIP created successfully")
				404 ("The containerName not found")
				503 ("Load balancer service is unavailable")
				409 ("VIP already exists")
				415 ("Invalid input data")

Update VIP	/one/nb/v2/lb/{container-name}/update/vip	PUT	{
			"name":"",
			"poolname":""
			}
				201 ("VIP updated successfully")
				404 ("The containerName not found")
				503 ("VIP not found")
				404 ("Pool not found")
				405 ("Pool already attached to the VIP")
				415 ("Invalid input name")

Delete VIP	/one/nb/v2/lb/{container-name}/delete/vip/{vip-name}	DELETE		200 ("VIP deleted successfully") 404 ("The containerName not found") 503 ("Load balancer service is unavailable") 404 ("VIP not found") 500 ("Failed to delete VIP")

Create pool member	/one/nb/v2/lb/{container-name}/create/poolmember	POST	{
			"name":"",
			"ip":"ip in (xxx.xxx.xxx.xxx) format",
			"poolname":"existing pool name"
			}
				201 ("Pool member created successfully")
				404 ("The containerName not found")
				503 ("Load balancer service is unavailable")
				404 ("Pool not found")
				409 ("Pool member already exists")
				415 ("Invalid input data")

Delete pool member	/one/nb/v2/lb/{container-name}/delete/poolmember/{pool-member-name}/{pool-name}	DELETE		200 ("Pool member

				deleted successfully") 404 ("The containerName not found") 503 ("Load balancer service is unavailable") 404 ("Pool member not found") 404 ("Pool not found")

3.6.3.2 使用负载均衡

运行控制器，确保 samples.loadbalancer 和 samples.loadbalancer.northbound 模块都被加载。

运行 mininet，连接到控制器，例如创建包含 16 台主机（10.0.0.1~10.0.0.16/8）的树状拓扑。

```
mn --topo=tree，2，4 --controller=remote，ip=<Host IP where controller is running>，port=6633
```

在控制器的 web ui 中添加网关（同一网段的未占用的 ip），之后各个主机可以互通。

创建轮询策略的负载均衡服务。

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://<Controller_IP>:8080/one/nb/v2/lb/default/create/pool -d '{"name":"PoolRR"，"lbmethod":"roundrobin"}'
```

创建 VIP 10.0.0.20。

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://Controller_IP:8080/one/nb/v2/lb/default/create/vip -d '{"name":"VIP-RR"，"ip":"10.0.0.20"，"protocol":"TCP"，"port":"5550"}'
```

添加 10.0.0.2 和 10.0.0.3 到池中。

```
curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
                http://Controller_IP:8080/one/nb/v2/lb/default/create/poolmember -d '{"name":"PM2"，"ip":"10.0.0.2"，"poolname":"PoolRR"}'

curl --user "admin":"admin" -H "Accept: application/json" -H "Content-type: application/json" -X POST
```

```
http://Controller_IP:8080/one/nb/v2/lb/default/create/poolmember -d '{"name":"PM3"，"ip":"10.0.0.3"，"poolname":"PoolRR"}'
```

因为 VIP 在网络中实际并不存在，无法解析 ARP 请求，因此需要手动添加对象的表项。

在 h1 上启动 xterm，添加 VIP 的 mac 表项。

```
arp -s 10.0.0.20 00:00:10:00:00:20
```

在 h2 到 h4 上启动 xterm，开启 iperf 服务端，例如监听 5550 端口（iperf –s –p 5550）。

在 h1 上启动 iperf 发送请求到 VIP（iperf -c 10.0.0.20 -p 5550）。

此时请求被发送到了 h2，结束后再次发送请求，则被发送到了 h3。

可以通过下面的命令来删除池中的成员

```
curl --user "admin":"admin"  -H "Accept: application/json" -H "Content-type: application/json" -X DELETE
                                                    http://Controller_IP:8080/one/nb/v2/lb/default/delete/poolmember/PM2/PoolRR'
```

删除 VIP。

```
curl --user "admin":"admin"  -H "Accept: application/json" -H "Content-type: application/json" -X DELETE
                                                    http://Controller_IP:8080/one/nb/v2/lb/default/delete/vip/VIP-RR
```

删除资源池。

```
curl --user "admin":"admin"  -H "Accept: application/json" -H "Content-type: application/json" -X DELETE
                                                    http://Controller_IP:8080/one/nb/v2/lb/default/delete/pool/PoolRR
```

3.7 库函数

3.7.1 C 客户端库

C 模块能生成 ANSI-C 兼容的 C 代码，可以跟 libxml2 结合，（解）序列化 XML 为 REST 资源。

生成的 C 代码依赖 XML Reader API 和 XML Writer API，以及<time.h>、<string.h>和<stdlib.h>等标准的 C 库。

REST XML 例子：

```
#include <full.c>
 //...

 xmlTextReaderPtr reader = ...; //set up the reader to the url.
 full_ns0_edgeProps *response_element = ...;
 response_element = xml_read_full_ns0_edgeProps(reader);
```

```
//handle the response as needed...

//free the full_ns0_edgeProps
free_full_ns0_edgeProps(response_element);
```

3.7.2 .NET 客户端库

.NET 客户端库定义了跟 XML 相互转化的类。例子：

```
//read a resource from a REST url
Uri uri = new Uri(...);

XmlSerializer s = new XmlSerializer(
  typeof( EdgeProps )
);

  //Create the request object
WebRequest req = WebRequest.Create(uri);
WebResponse resp = req.GetResponse();
Stream stream = resp.GetResponseStream();
TextReader r = new StreamReader( stream );

EdgeProps order = (EdgeProps) s.Deserialize( r );

//handle the result as needed...
```

3.7.3 Java 客户端库

Java 客户端库用于访问应用的 Web 服务 API。
JAX-WS 客户端库用于提供可以利用 JAXB 跟 XML 之间相互转化的 java 对象。
Raw JAXB 的 REST 代码例子：

```
java.net.URL url = new java.net.URL(baseURL + "/{containerName}");
JAXBContext context = JAXBContext.newInstance( EdgeProps.class );
java.net.URLConnection connection = url.openConnection();
connection.connect();

Unmarshaller unmarshaller = context.createUnmarshaller();
```

```
 EdgeProps result = (EdgeProps) unmarshaller.unmarshal( connection.getInputS
tream() );
 //handle the result as needed...
```

Jersey 客户端的 REST 代码例子：

```
</nowiki>
com.sun.jersey.api.client.Client client = com.sun.jersey.api.client.Client.create();

EdgeProps result = client.resource(baseUrl + "/{containerName}")

  .get(EdgeProps.class);

//handle the result as needed...
</nowiki>
```

3.7.4 Java JSON 客户端库

提供了跟 jackson 之间转化的 java 对象。代码例子：

```
java.net.URL url = new java.net.URL(baseURL + "/{containerName}");
 ObjectMapper mapper = new ObjectMapper();
 java.net.URLConnection connection = url.openConnection();
 connection.connect();

 EdgeProps result = (EdgeProps) mapper.readValue( connection.getInputStrea
m()， EdgeProps.class );
 //handle the result as needed...
```

3.7.5 Objective C 客户端库

Objective C 模块必须生成能用于 libxml2 的 Objective C 类和序列化相关函数。

生成的 Objective C 源代码依赖于 XML Reader API 和 XML Writer API，和基本的 OpenStep 基础类。例子：

```
#import <full.h>
 //...

 FULLNS0EdgeProps *responseElement;
 NSData *responseData; //data holding the XML from the response.
 NSURL *baseURL = ...; //the base url including the host and subpath.
```

```
 NSURL *url = [NSURL URLWithString: @"/{containerName}" relativeToURL: bas
eURL];
 NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:
url];
 NSURLResponse *response = nil;
 NSError *error = NULL;
 [request setHTTPMethod: @"GET"];

 //this example uses a synchronous request,
 //but you'll probably want to use an asynchronous call
 responseData = [NSURLConnection sendSynchronousRequest:request returnin
gResponse:&response  error:&error];
 FULLNS0EdgeProps *responseElement = [FULLNS0EdgeProps readFromXML: r
esponseData];
 [responseElement retain];

 //handle the response as needed...
```

3.8 REST 调用和认证

REST API 包括多个模块。同时提供了 HTTP Digest 认证的 REST 认证。管理员用户可以通过 web 来管理用户。以后将支持 HTTPs，并把 REST API 迁移到 HTTP Basic。

3.8.1 Topology REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/topology/target/site/wsdocs/index.html。

3.8.2 Host Tracker REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/hosttracker/target/site/wsdocs/index.html。

3.8.3 Flow Programmer REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/flowprogrammer/target/site/wsdocs/index.html。

3.8.4 Static Routing REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/staticrouting/target/site/wsdocs/index.html。

3.8.5 Statistics REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/stati

stics/target/site/wsdocs/index.html。

3.8.6 Subnets REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/subnets/target/site/wsdocs/index.html。

3.8.7 Switch Manager REST APIs

https://jenkins.opendaylight.org/controller/job/controller-merge/ws/opendaylight/northbound/switchmanager/target/site/wsdocs/index.html。

3.9 Java API

参考自动生成的 javadoc：https://jenkins.opendaylight.org/controller/job/controller-merge/lastSuccessfulBuild/artifact/opendaylight/distribution/opendaylight/target/apidocs/index.html。

3.10 拓扑

Open Daylight Controller 提供了对物理网络的集中式的逻辑拓扑。并且可以更改转发的规则。控制器实现拓扑是基于 LLDP 消息。

Web UI 中可以看到连接到控制器的交换机信息以及主机的信息。

所有拓扑信息是由 Topology Manager 来维护的。

3.11 集成测试

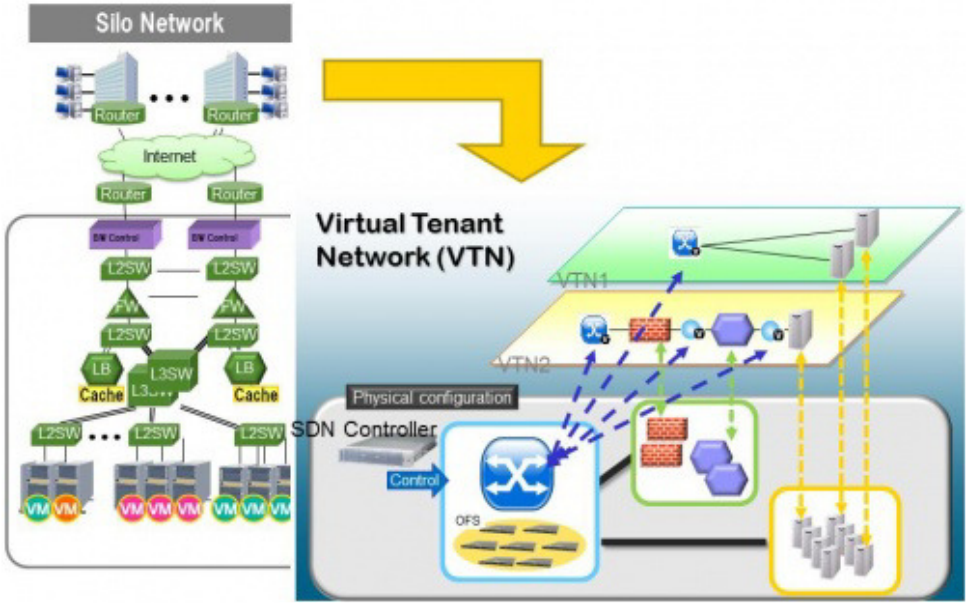
第4章 VTN

Virtual Tenant Network 项目是 ODP 中的一个重要项目，主要负责在 SDN 控制器上提供对多租户虚拟网络的支持。利用该项目，可以很好的让 ODP 支持 OpenStack。

4.1 概念

4.1.1 场景

所解决的场景就是现在数据中心网络的场景。不同租户需要运行不同的网络配置和应用程序，这些需要在同一个物理网络中实现，并且实现资源的共用。如果不使用虚拟网络，则需要在同一个物理网络中针对不同租户分别配置和管理，情况将变得十分复杂。



图表 13 多租户虚拟网络场景

而在支持虚拟网络的情况下，VTN 提供一个逻辑的抽象层，租户的隔离是在逻辑层上进行的，底下的物理网络不需要进行太多的改动。对租户来说，看到的都是抽象网络拓扑，不需要关心具体的物理拓扑结构和具体的网络参数。图表 13 中给出了一个场景示例。

而 VTN 应用，则负责完成从用户的抽象网络到底下物理网络的映射，并且通过 SDN 协议来负责各个交换节点上的规则配置。

4.1.2 术语

VTN 的基本元素包括虚拟节点（vBridge、vRouter）、虚拟接口和虚拟链路。通过虚拟链路将虚拟节点上的虚拟接口连接起来，即形成一张虚拟网络。

4.1.2.1 虚拟节点

vBridge：代表一个 L2 交换机，实现上可以是同一个物理 VLAN 的所有或部分交换机。
vRouter：代表一个路由器。
vTep：代表一个隧道的端点。
vTunnel：一个隧道。

vBypass：受控网络之间的连通性。

4.1.2.2 虚拟接口

interface：表示一个虚拟节点上的一个接口。

4.1.2.3 虚拟链路

vLink：虚拟接口之间的 L1 通路。

4.2 物理网映射

如何完成抽象网络和物理网之间的映射是 VTN 的核心问题。

具体来说，当一个物理网上的交换机收到网包，VTN 需要知道是哪个逻辑网，反过来完成转发，还要从逻辑网告知物理网。

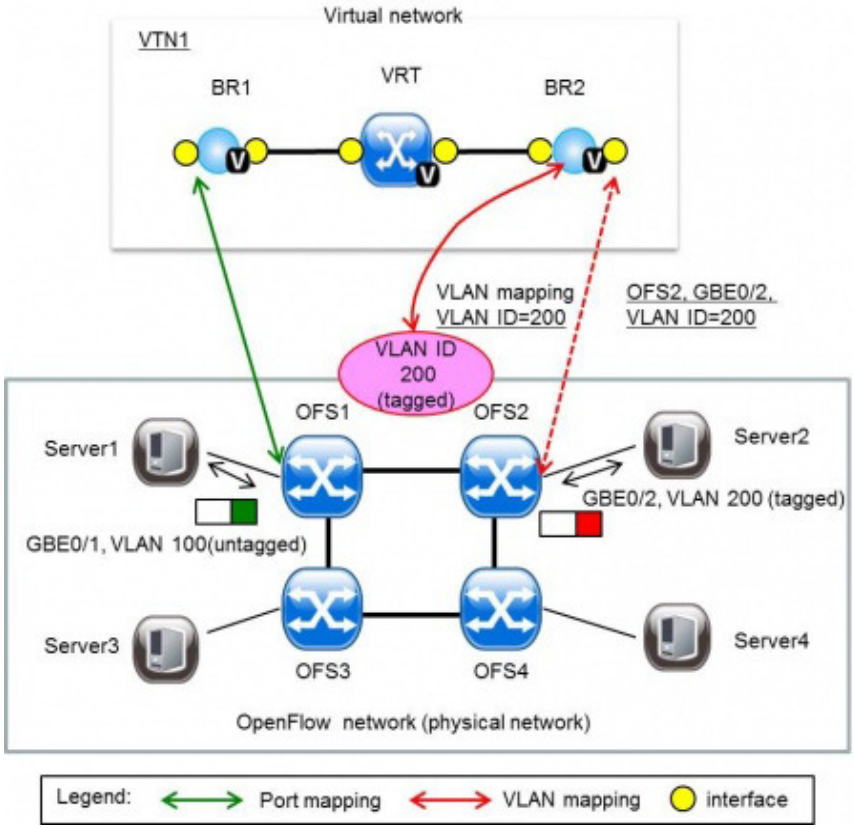
映射包括三种方法：

端口映射，利用网包到达的物理交换机 id、端口 id 和网包的 VLAN id 来映射到一个 vBridge 的一个接口上。

VLAN 映射，利用网包的 VLAN id 来映射到一个 vBridge。或者利用利用到达物理交换机 id 和网包 VLAN id 来映射一个特定交换机的资源到一个 vBridge。

MAC 映射，利用网包的 mac 地址把物理资源映射到一个 vBridge 的一个接口。

此外，VTN 可以学习挂载到交换机上的终端信息，并保存 MAC 地址和 VLAN id 到交换机的端口的关联信息。当终端断开后，经过超时机制会丢弃该信息。



图表 14 VTN mapping 示例

图表 14 中给出了一个映射的例子。该例中，BR1 的一个接口跟 OFS1 的一个接口相映射，而 VLAN id 为 200 的网络，全部被映射为 BR2。

4.3 功能

4.3.1 vBridge

即常规的 L2 交换机功能，根据目的 MAC 进行查表后转发到某个端口。如果不存在表项，则进行 flooding。

收到网包时，学习源 mac，并且利用超时机制来清除断开的主机。

同时，支持静态配置 mac 信息。

4.3.2 vRouter

在 vBridge 之间传输 IP 网包，支持路由和 ARP。

当一个 IP 地址注册到某个路由器的虚接口的时候，会注册对应的路由信息。支持静态配置。

同时，为每一个路由域维护一张 arp 表，绑定目标 IP、MAC 地址到虚拟接口上。通过超时机制来删除过期信息。支持静态配置。

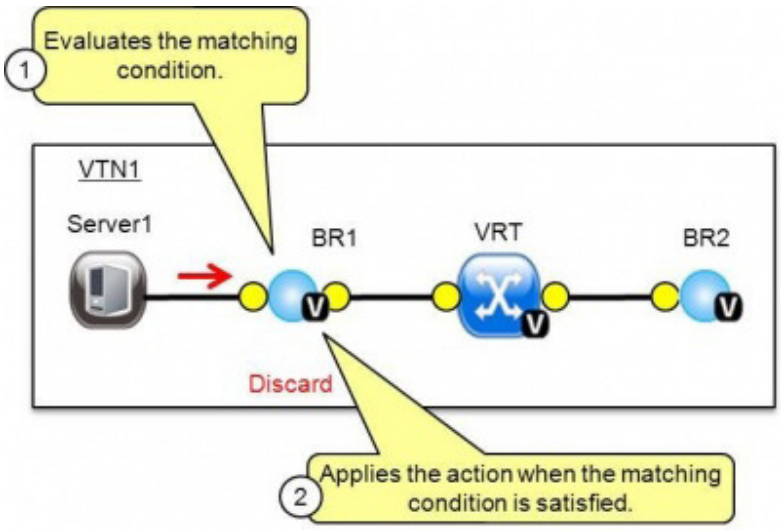
支持 DHCP 的传递。

4.3.3 流过滤

类似 ACL。对网包进行简单的允许或禁止通过。规则的配置可以部署到某个接口上。

支持的过滤域包括：Source MAC address， Destination MAC address， MAC ether type， VLAN Priority， Source IP address， Destination IP address， DSCP， IP Protocol， TCP/UDP source port， TCP/UDP destination port， ICMP type， ICMP code 等。

支持的行动包括 Pass，Drop 或 Redirection（包括透明发送过去和修改了 mac 路由过去）。



图表 15 VTN 流过滤示例

图表 15 中给出了一个示例。在 BR1 上进行过滤。

4.3.4 多控制器合作

VTN 支持多个控制器之间的合作配置，支持动态添加控制器和删除控制器。

例如，每个控制器上配置一个 VTN，但集成为一个单独的 VTN 提供统一的策略。

VTN 支持同时管理 OpenFlow 的网络和 Overlay 的网络。

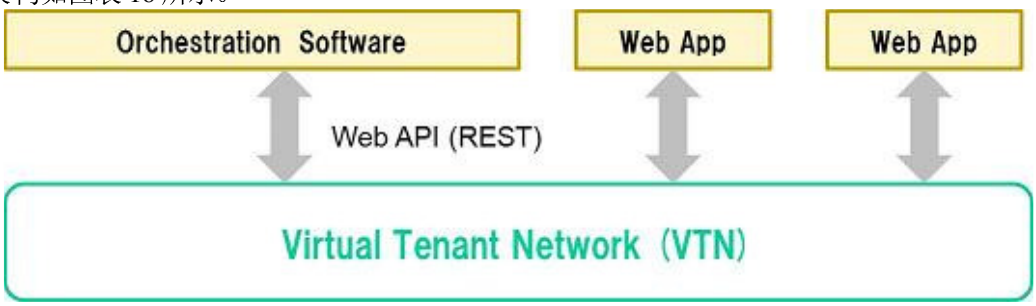
4.3.4.1.1 OpenFlow 网络和 L2/L3 网络协作

如果网络中混合有支持 OpenFlow 的交换机和传统的 L2/L3 交换机，在 VTN 中传统交换设备作为 vBypass 存在。注意在 vBypass 上无法配置过滤策略。

4.3.5 北向 API

VTN 提供了 rest API，因此用户可以通过 web 操作来对 VTN 资源进行管理。支持 Json 和 Xml 格式。

架构如图表 16 所示。



图表 16 VTN 北向 API

支持的操作如表格 2 所示：

表格 2 北向 API 支持操作

Resources	GET	POST	PUT	DELETE
VTN	Yes	Yes	Yes	Yes
vBridge	Yes	Yes	Yes	Yes
vRouter	Yes	Yes	Yes	Yes
vTep	Yes	Yes	Yes	Yes
vTunnel	Yes	Yes	Yes	Yes
vBypass	Yes	Yes	Yes	Yes
vLink	Yes	Yes	Yes	Yes
Interface	Yes	Yes	Yes	Yes

Port map	Yes	No	Yes	Yes
Vlan map	Yes	Yes	Yes	Yes
Flowfilter (ACL/redirect)	Yes	Yes	Yes	Yes
Controller information	Yes	Yes	Yes	Yes
Physical topology information	Yes	No	No	No
Alarm information	Yes	No	No	No

4.3.5.1 北向 API 例子

创建 VTN

```
# curl -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: PASSWORD' -H 'ipaddr: 127.0.0.1' \
  -d '{"vtn":{"vtn_name":"VTN1"}}' http://172.1.0.1:8080/vtn-webapi/vtns.json
```

创建控制器信息

```
# curl -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: PASSWORD' -H 'ipaddr: 127.0.0.1' \
  -d '{"controller": {"controller_id":"CONTROLLER1","ipaddr":"172.1.0.1","type":"pfc","username":"root", \
  "password":"PASSWORD","version":"5.0"}}'  http://172.1.0.1:8080/vtn-webapi/controllers.json
```

创建 vBridge

```
# curl -X POST -H 'content-type: application/json' -H 'username: admin' -H 'password: PASSOWRD' -H 'ipaddr: 127.0.0.1' \
  -d '{"vbridge":{"vbr_name":"VBR1","controller_id": "CONTROLLER1","domain_id": "(DEFAULT)"}}' \
  http://172.1.0.1:8080/vtn-webapi/vtns/VTN1/vbridges.json
```

创建接口，连接到 vBridge 上的一个终端上

```
# curl -X POST -H 'content-type: application/json' -H 'username: admin' -H 'pas
sword: PASSWORD' -H 'ipaddr: 127.0.0.1' \
 -d '{"interface":{"if_name":"IF1"}}' http://172.1.0.1:8080/vtn-webapi/vtns/VTN1/
vbridges/VBR1/interfaces.json
```

4.4 安装

VTN Coordinator 运行在 ODC（OpenDaylight Controller）之外，而 VTN Manager 作为一个 bundle，运行在控制器中。因此，一般推荐分别安装两者到不同机器。

另外，VTN Coordinator 的 web 端口默认是 8080，与 ODC 冲突，需要进行修改：

```
[root@tsukemen conf]# pwd
/usr/share/java/apache-tomcat-7.0.39/conf
[root@tsukemen conf]# diff -u server.xml.org server.xml
--- server.xml.org      2013-08-08 09:53:53.748971829 +0900
+++ server.xml  2013-08-08 09:15:39.012970589 +0900
@@ -68,13 +68,13 @@
        APR (HTTP/AJP) Connector: /docs/apr.html
        Define a non-SSL HTTP/1.1 Connector on port 8080
    -->
-    <Connector port="8080" protocol="HTTP/1.1"
+    <Connector port="8081" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
    <!-- A "Connector" using the shared thread pool-->
    <!--
    <Connector executor="tomcatThreadPool"
-              port="8080" protocol="HTTP/1.1"
+              port="8081" protocol="HTTP/1.1"
               connectionTimeout="20000"
               redirectPort="8443" />
    -->
```

4.4.1 VTN Coordinator 安装运行

4.4.1.1 准备

以 redhat os 为例。需要 RHEL6.1 或更新的版本。

安装必要的程序包。

```
yum install make glibc-devel gcc gcc-c++ boost-devel openssl-devel \
```

```
ant perl-ExtUtils-MakeMaker unixODBC-devel perl-Digest-SHA uuid libxslt libcurl libcurl-devel git
```

安装 JDK7，并且设置 JAVA_HOME 环境变量。

```
yum install java-1.7.0-openjdk-devel
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk.x86_64
```

从 http://yum.postgresql.org/9.1/redhat/rhel-6-x86_64 安装 postgreSQL

```
postgresql91-libs-9.1.9-1PGDG.rhel6.x86_64.rpm
postgresql91-9.1.9-1PGDG.rhel6.x86_64.rpm
postgresql91-server-9.1.9-1PGDG.rhel6.x86_64.rpm
postgresql91-contrib-9.1.9-1PGDG.rhel6.x86_64.rpm
postgresql91-odbc-09.00.0310-1PGDG.rhel6.x86_64.rpm
```

安装 maven。

安装 gtest-devel，json-c。

```
wget http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm
rpm -Uvh epel-release-6-8.noarch.rpm
yum install gtest-devel json-c json-c-devel
```

4.4.1.2 编译

下载代码。

```
git clone ssh://<username>@git.opendaylight.org:29418/vtn.git
```

编译，并安装 VTN Coordinator

```
cd vtn/coordinator
mvn -f dist/pom.xml package
sudo make install
```

4.4.1.3 运行

安装 Tomcat，从 http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.39/bin/apache-tomcat-7.0.39.tar.gz 下载后解压到/usr/share/java。

```
tar zxvf apache-tomcat-7.0.39.tar.gz -C /usr/share/java
```

配置 Tomcat。

创建软连接

```
ln -s /usr/local/vtn/tomcat/webapps/vtn-webapi /usr/share/java/apache-tomcat-7.0.39/webapps/vtn-webapi
```

添加下面的路径到/usr/share/java/apache-tomcat-7.0.39/conf/catalina.properties 文件的 common.loader。

```
/usr/local/vtn/tomcat/lib,/usr/local/vtn/tomcat/lib/*.jar
```

添加下面的路径到/usr/share/java/apache-tomcat-7.0.39/conf/catalina.properties 文件的 shared.loader。

```
/usr/local/vtn/tomcat/shared/lib/*.jar
```

添加下面的路径到/usr/share/java/apache-tomcat-7.0.39/conf/server.xml 文件的<Server>。

```
<Listener className="org.opendaylight.vtn.tomcat.server.StateListener" />
```

配置 DB。

```
/usr/local/vtn/sbin/db_setup
```

4.4.1.3.1 启动和停止

启动 VTN Coordinator 和 Tomcat：

```
/usr/local/vtn/bin/vtn_start
/usr/share/java/apache-tomcat-7.0.39/bin/catalina.sh start
```

如果要停止，执行下面的命令：

```
/usr/share/java/apache-tomcat-7.0.39/bin/catalina.sh stop
/usr/local/vtn/bin/vtn_stop
```

4.4.1.3.2 WebAPI

启动 VTN Coordinator 后，通过下面的命令可以获取版本信息。

```
$ curl -X GET -H 'content-type: application/json' -H 'username: admin' -H 'pass
word: adminpass' -H 'ipaddr:127.0.0.1' http://127.0.0.1:8080/vtn-webapi/api_ver
sion.json
{"api_version":{"version":"V1.0"}}
```

4.4.2 VTN Manager 安装运行

4.4.2.1 准备

VTN Manager 作为 ODC 的一个 bundle，因此只要能满足 ODC 的运行环境即可。

4.4.2.2 编译

下载最新的代码：

```
git clone ssh://<username>@git.opendaylight.org:29418/vtn.git
```

编译代码：

```
cd ${VTN_DIR}
mvn -f manager/dist/pom.xml install
```

4.4.2.3 运行

运行命令为：

```
cd  ${VTN_DIR}/manager/dist/target/distribution.vtn-manager-0.1.0-SNAPSHO
T-osgipackage/opendaylight
./run.sh
```

4.4.2.3.1 REST API

创建虚拟租户网络：

```
curl --user "admin":"admin" -H "Accept: application/json" -H \
"Content-type: application/json" -X POST \
http://localhost:8080/controller/nb/v2/vtn/default/vtns/Tenant1 \
-d '{"description": "My First Virtual Tenant Network"}'
```

查看所有租户：

```
curl --user "admin":"admin" -H "Accept: application/json" -H \
"Content-type: application/json" -X GET \
http://localhost:8080/controller/nb/v2/vtn/default/vtns
```

4.4.2.3.2 使用 mininet

使用 mininet 来运行多个控制器，多个控制器需要在 VTN Coordinator 的管理下，因此使用 mininet 的 multitree.py 脚本来模拟多个控制器的网络。这个脚本将启动 6 台交换机，和两个控制器，各负责管理 3 台交换机。

编辑脚本中的 ControllerAddress，之后执行脚本：

```
% sudo python multitree.py
```

该脚本的源代码为：

```
#!/usr/bin/python

"""
Run Mininet network using tree topology per remote controller.
"""

from mininet.cli import CLI
from mininet.log import info, setLogLevel
from mininet.net import Mininet
from mininet.node import Host, OVSKernelSwitch, RemoteController
from mininet.topo import Topo

TreeDepth = 2
FanOut = 2
ControllerAddress = ["192.168.0.180", "192.168.0.181"]

class MultiTreeTopo(Topo):
    """Topology for multiple tree network using remote controllers.
    A tree network is assigned to a remote controller."""
```

```
def __init__(self):
    Topo.__init__(self)

    self.hostSize = 1
    self.switchSize = 1
    self.treeSwitches = []

    prev = None
    for cidx in range(len(ControllerAddress)):
        switches = []
        self.treeSwitches.append(switches)
        root = self.addTree(switches, TreeDepth, FanOut)
        if prev:
            self.addLink(prev, root)
        prev = root

def addTree(self, switches, depth, fanout):
    """Add a tree node."""
    if depth > 0:
        node = self.addSwitch('s%u' % self.switchSize)
        self.switchSize += 1
        switches.append(node)
        for i in range(fanout):
            child = self.addTree(switches, depth - 1, fanout)
            self.addLink(node, child)
    else:
        node = self.addHost('h%u' % self.hostSize)
        self.hostSize += 1

    return node

def start(self, net):
    """Start all controllers and switches in the network."""
```

```
        cidx = 0
        for c in net.controllers:
            info("*** Starting controller: %s\n" % c)
            info("    + Starting switches ... ")
            switches = self.treeSwitches[cidx]
            for sname in switches:
                s = net.getNodeByName(sname)
                info(" %s" % s)
                s.start([c])
            cidx += 1
            info("\n")

        self.treeSwitches = None

class MultiTreeNet(Mininet):
    """Mininet network environment with multiple tree network using remote
    controllers."""

    def __init__(self, **args):
        args['topo'] = MultiTreeTopo()
        args['switch'] = OVSKernelSwitch
        args['controller'] = RemoteController
        args['build'] = False
        Mininet.__init__(self, **args)

        idx = 1
        for addr in ControllerAddress:
            name = 'c%d' % idx
            info('*** Creating remote controller: %s (%s)\n' % (name, addr))
            self.addController(name, ip=addr, port=6633)
            idx = idx + 1

    def start(self):
        "Start controller and switches."
        if not self.built:
```

```
            self.build()

        self.topo.start(self)

if __name__ == '__main__':
    setLogLevel('info')  # for CLI output
    net = MultiTreeNet()
    net.build()

    print "*** Starting network"
    net.start()

    print "*** Running CLI"
    CLI(net)

    print "*** Stopping network"
    net.stop()
```

4.5 虚拟化版本

ODC 的虚拟化版本中包括了 vtn 的 manager，因此，我们可以通过运行虚拟化版本来使用 vtn。

直接下载完整版 ODC，并运行控制器，并制定虚拟化版本参数。

```
* cd opendaylight
 * ./run.sh -virt vtn
```

4.5.1 安装 VTN Coordinator

将 vtn coordinator 的压缩包作为虚拟化版本的外部 app 使用。

```
cd opendaylight/externalapps
tar -C / -jxvf org.opendaylight.vtn.distribution.vtn-coordinator-5.0.0.0-<date_time_build_ref>-bin.tar.bz2
```

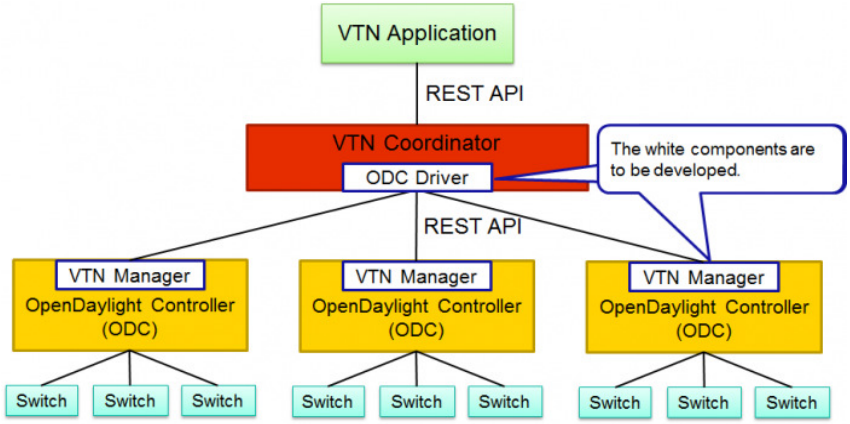
该命令将安装 Coordinator 到/usr/local/vtn 路径。

4.5.2 运行 VTN Coordinator

参考4.4.1VTN Coordinator 安装运行。

4.6 实现

4.6.1 整体架构



图表 17 VTN 整体架构

图表 17 给出了 VTN 项目的整体架构。

主要包括两大部分：VTN Coordinator 和 VTN Manager。

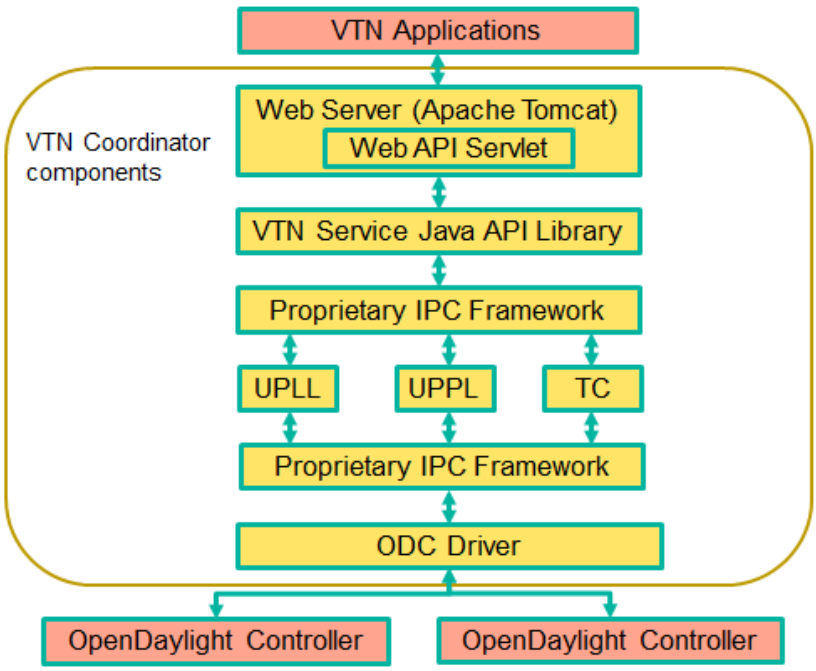
其中 VTN Coordinator 为 VTN 的应用程序提供北向的 REST API 支持，同时负责组织协调底下的若干控制器。

VTN Manager 作为控制器的一个 bundle，运行在 OSGi 框架下，为 VTN Coordinator 提供具体的虚拟化网络功能支持。

4.6.2 VTN Manager

4.6.3 VTN Coordinator

图表 18 描绘了 VTN Coordinator 的整体架构。



图表 18 VTN Coordinator 整体架构

包括的主要组件有

VTN API：提供 VTN 的北向 Web API

TC：两步提交模块

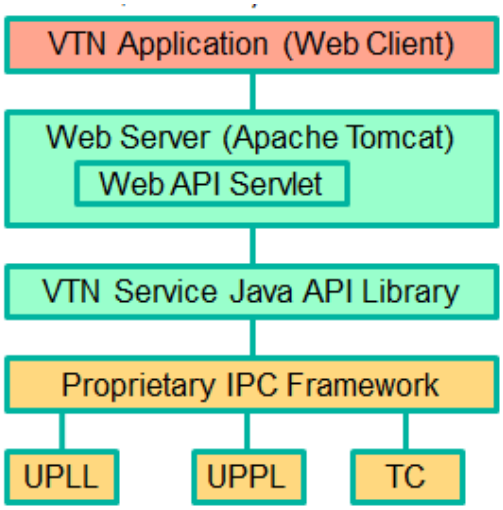
UPPL：物理网的管控模块

UPLL：虚拟网的管控模块

ODC Driver：控制器接口模块

4.6.3.1 VTN API

架构如图表 19 所示。



图表 19 VTN API 架构

包括两个子模块：Web Server 和 VTN Service Java API 库。

前者处理 VTN 的应用程序发来的 REST 请求，并将这些请求转化为对应的 Java API，该

模块的主要功能包括：通过启动脚本 catalina.sh 启动；VTN 应用程序通过 Json 或 XML 格式发送 HTTP 请求；创建一个 session，并获取读写锁；调用对应的 VTN Service Java API；返回响应到 VTN 应用。

后者提供了调用底层 Java 模块的 API，主要功能包括：创建一个面向底层的 IPC 客户端 session；将请求转化为 IPC 框架格式；调用底层的 API（包括 UPPL，UPLL，TC 的 API 等）；从底层返回应答给 Web 服务器。

表格 3 VTN Coordinator Web Server 中的类

Class Name	Description
InitManager	It is a Singleton class for executing the acquisition of configuration information from properties file, log initialization, initialization of VTN Service Java API. Executed by init() of VtnServiceWebAPIServlet.
ConfigurationManager	Class to maintain the configuration information acquired from properties file.
VtnServiceCommonUtil	Utility class.
VtnServiceWebUtil	Utility class.
VtnServiceWebAPIServlet	Receives HTTP request from VTN Application and calls the method of corresponding VtnServiceWebAPIHandler. Inherits class HttpServlet, and overrides doGet(), doPut(), doDelete(), doPost().
VtnServiceWebAPIHandler	Creates JsonObject(com.google.gson) from HTTP request, and calls method of corresponding VtnServiceWebAPIController.
VtnServiceWebAPIController	Creates RestResource() class and calls UPLL API/UPPL API through Java API. At the time of calling UPLL API/UPPL API, performs the creation/deletion of session, acquisition/release of configuration mode, acquisition/release of read lock by TC API through Java API.
DataConverter	Class to covert from HTTP request to JsonObject and from JsonXML to JSON.

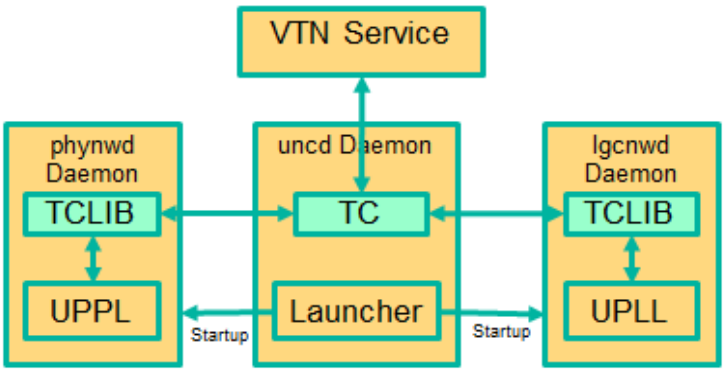
表格 4 VTN Coordinator Service Java API 库中的类

Class Name	Description
VtnServiceInitManager	It is a Singleton class for executing the acquisition of configuration information from properties file, log initialization.

	Executed by init() of Web API Servlet.
VtnServiceConfiguration	Class to maintain the configuration information acquired from properties file.
IpcConnPool	Class that mains Connection pool of IPC.
IpcChannelConnection	Class that mains Connections of IPC.
RestResource	The class that will be interface for Web API Servlet. Implementation of Interface VtnServiceResource.
AnnotationReflect	Performs the mapping of path filed value of RestRsource class and xxxResource class.
xxxResource	The class that is created according to the path filed value of RestResource. (vtnResource, VBridgeResource etc) Inherits abstract class AbstractResource.
xxxResourceValidator CommonValidator	The class that performs the appropriateness check of values specified in the path, query, request field of RestResource class.
IpcPhysicalResponseFactory	The class to create JsonObject from the response received from UPPL.
IpcRequestProcessor	Sends request to UPLL/UPPL through proprietary IPC Framework. UPLL API and UPPL API are the API that are implemented on proprietary IPC Framework, and request/response is defined by special interface called as Key Interface.
IpcRequestPacket	The class that maintains the request to be sent to UPLL/UPPL.
IpcStructFactory	The class to create Key Structure and Value Structure that will be included in the Request to be sent to UPLL /UPPL.

4.6.3.2 TC

提供两步提交协调功能，包括两个子组件：Transaction Coordinator 和 Transaction Coordinator Library。



图表 20 VTN Coordinator TC 模块组件

其中，前者支持功能包括：启动时候从 uncd daemon 启动；响应 VTN 中的两步提交操作；在提交和审计操作时从 VTN 服务接收请求；通过 IPC 框架来调用底层的 TCLIB API（UPLL、UPPL 或 ODC Driver API）。

后者作为 UPLL，UPPL 和 ODC Driver daemon 的一个模块被加载；负责处理从 TC 发给 daemon 的消息；这些 daemon 将安装他们的句柄到 TCLIB，这些句柄在收到 TC 的消息时被调用。

表格 5 VTN Coordinator TC 模块的类

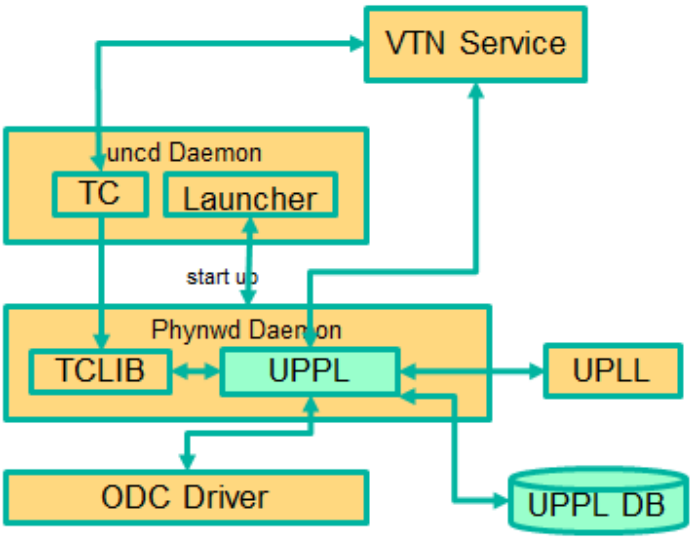
Class Name	Description
TcModule	Main interface which offers the services to VTN Service library. It also handles state transitions.
TcOperations	Base class that services every operation request in TC.
TcMsg	The message to be sent for every operation has different characteristics based on the type of message. This base class will provide methods to handle different types of messages to the intended recipients.
TcLock	The exclusion control class, an object of TcLock is contained in TcModule and used for every operation.
TcDbHandler	Utility class for TC database operations.
TcTaskqUtil	Utility class for taskq used in TC for driver triggered audit and read operations.

表格 6 VTN Coordinator TCLIB 模块的类

Class Name	Description
TcLibModule	Main class which handles requests from TC module.
TcLibInterface	Abstract class which every module implements to interact with TC module. Member of TcLibModule.
TcLiBMsgUtil	Internal utility class for extracting session attributes of every request from TC.

4.6.3.3 UPPL

提供对物理网络的管控，如图表 21 所示。



图表 21 VTN Coordinator UPPL 模块示意

启动是通过 phynwd daemon；通过 IPC 框架跟 TC、UPLL 和 ODC Driver 进行交互；从 VTN Service 获取各种 CRUD 请求；在外部数据库中维护启动信息和配置、状态信息；在 TC 的指示下进行 setup/commit/abort 操作；通过 ODC Driver 连接到南向的控制器；利用控制器的通知创建物理拓扑；通知 UPLL 关于控制器添加、删除等操作和物理拓扑的改变。

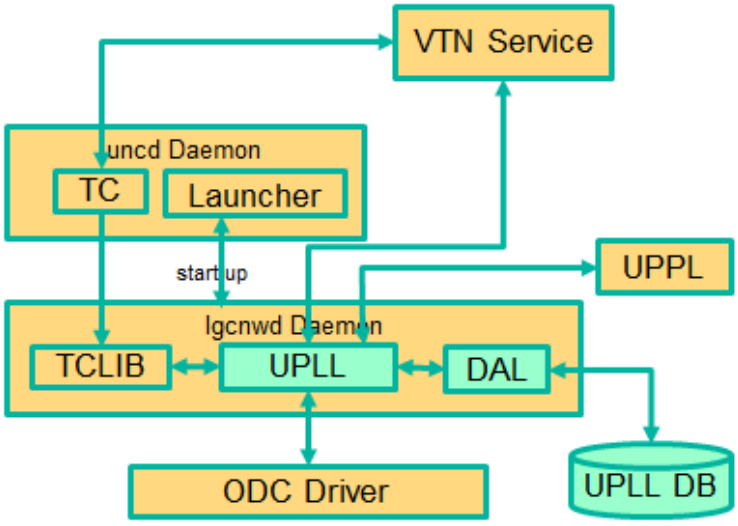
表格 7 VTN Coordinator UPPL 模块中的类

Class Name	Description
PhysicalLayer	It's a singleton class which will instantiate other UPPL's classes. This class will be inherited from base module in order to use the Core features and IPC service handlers.
PhysicalCore	Class that is responsible for processing requests from TC. It also processes the configuration and capability file. It's also responsible for sending alarm to node manager. It's responsible for receiving requests from north bound.
IPCConnectionManager	It's responsible for processing the requests received via IPC framework. It contains separate classes to process request from VTN service, UPLL and ODC Driver.
ODBCManager	It is a singleton class which performs all database services.
InternalTransactionCoordinator	It is responsible for parsing the IPC structures and forward it to the various request classes like ConfigurationRequest, ReadRequest,

	ImportRequest etc.
ConfigurationRequest	It is responsible to process the Create, Delete and Update operations received from VTN service.
ReadRequest	It is responsible to process all the read operations.
Kt_Base, Kt_State_Base and respective Kt classes	These classes perform the functionality required for individual key type.
TransactionRequest	It is responsible for performing the various functions required for each phase of the Transaction Request received from Transaction Coordinator during User Commit/Abort.
AuditRequest	It is responsible for performing functions related to audit request.
ImportRequest	It is responsible for performing functions related to import request.
SystemStateChangeRequest	It is responsible for performing functions when VTN coordinator state is moved to active or standby.
DBConfigurationRequest	It is responsible for processing various Database operations like Save/Clear/Abort

4.6.3.4 UPLL

提供对虚拟网的监控。包括 UPLL 和 DAL。
如图表 22 所示。



图表 22 UPLL 在架构中的位置

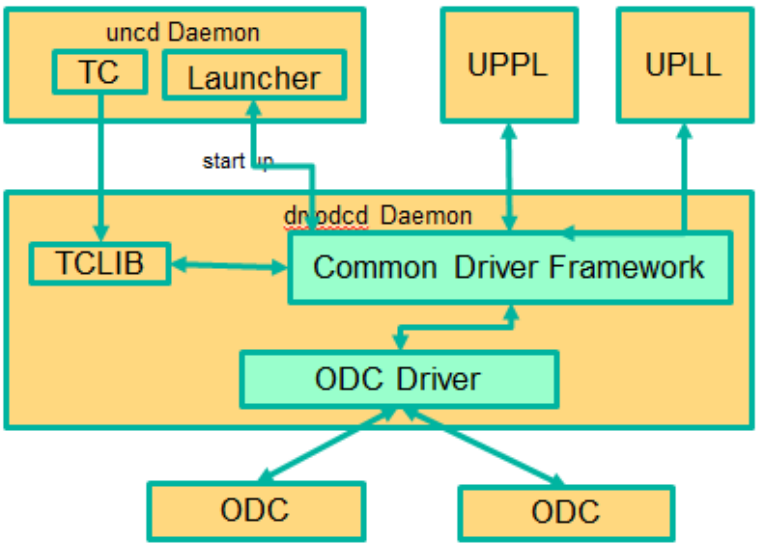
主要功能包括：在启动时候被 lgcnwd daemon 启动；通过 IPC 框架分别跟 TC、UPPL 和 ODC Driver 进行交互；通过 VTN 服务接受对虚拟网络的 CURD 请求；在外部数据库维护启动、运行配置和状态信息等数据；在 TC 指导下进行 Setup/Commit/Abort 等操作；通过 ODC Driver 来连接到南向的控制器；维护从控制器传来的虚拟网拓扑相关信息；为虚拟网配置提供审计和导入。

此外，DAl 模块实现了对数据库的抽象层。

4.6.3.5 ODC Driver

ODC Driver 模块提供监控虚拟网络和控制器的物理网络，在 VTN Coordinator 的启动时被启动，主要包括常规驱动框架（Common Driver Framework，CDF）和 ODC Driver。

如图表 23 所示。



图表 23 ODC Driver 在架构中位置

主要是面向 UPLL 和 UPPL，提供独立于控制器的一层。

功能包括：处理收到的 UPLL 和 UPPL 的消息；提供驱动接口来在控制器上执行命令；提供对不同类型控制器的支持；解析消息和出发驱动方法；为不同驱动提供接口来安装命令句柄；提供封装，简化 vote 和 commit 操作的操作；支持在多个控制器上的并行更新；可以扩展支持多个驱动模块。

CDF 是基于 vtndrvintf 和 vtncacheutil 实现的。

ODC Driver 模块实现了控制器管理和在控制器中提供虚拟网支持的接口。上层发来的请求将被转化为合适的 REST API 调用，然后发给控制器。转换 VTN 的操作为 VTN Manager 的命令。

ODC Driver 主要由 restjsonutil、odcdriver 两个子模块实现。前者是提供对 JSON 的解析处理；后者实现了 CDF 开放的接口，并为 ODC 注册驱动，同时使用 restjsonutil 来通信。

4.6.4 VTN Coordinator REST Reference

遵循 REST 的一般使用规则。

POST 为创建新资源，PUT 为更新，GET 为获取，DELETE 为删除。

请求头需要的信息如表格 8 所示。

表格 8 VTN Coordinator REST 头部信息

域	POST	PUT	GET	DELETE
username	Y	Y	Y	Y
password	Y	Y	Y	Y
Accept	N	N	N	N
Content-Type	Y	Y	N	N
Content-Length	Y	Y	N	N
Host	Y	Y	Y	Y