中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告
Undergraduate Experiment Report

题目 Title: **计算机网络 (II) 实验报告**

院 系
School (Department): 数据科学与计算机学院

专 业
Major: 网络空间安全

学生姓名
Student Name: 王广烁 金钰 李晨曦

学 号
Student No.: 18340165 18340081 18340087

时间：二〇二一年一月六日
Date: January 6th 2021

目录

插图目录

表格目录

第一章 背景

1.1 引言

SDN 代表了网络演进中的一种重大范式转变，其引入了网络基础设施创新的新步伐。其核心理念是希望应用软件可以参与对网络的控制管理，满足上层业务需求，通过自动化业务部署简化网络运维。或者说，只要网络硬件可以集中式软件管理，可编程化，控制转发层面分开，则可以认为这个网络是一个 SDN 网络。

SDN 的优势在于，其将网络由硬变软，提升工程师对网络的集中控制能力，最终提升网络对业务的服务、支持是能力。另外，SDN 的控制、配置等工作更加开放，用户可以自定义网络路由、包运输策略规则，从而更加灵活和智能。进行 SDN 改造后，基于其的可编程特性，工程师无需再对网络中每个节点的路由器反复进行配置，网络中的设备本身就是自动化连通的，只需要在使用时定义好简单的网络规则即可。当然，如果路由器自身内置的协议不符合用户的需求，可以通过编程的方式对其进行修改，以实现更好的数据交换性能。

1.2 SDN 应用与仿真分析

此部分将介绍我们实验中所使用到的软件、库、控制器的作用，以及配置及基本使用方法

1.2.1 SDN 网络仿真工具：Mininet

Mininet 是 SDN 仿真工具，可以创建包含主机，交换机，控制器和链路的虚拟网络。其能够为应用程序提供一个简单的网络测试平台；可以进行复杂的拓扑测试；拥有拓扑感知的 CLI 用于调试或运行相关网络的测试；支持任意自定义拓扑并且包括基本的参数化拓扑。

下面介绍 mininet 的安装工作

```
git clone git://github.com/mininet/mininet  # 下载 mininet
cd mininet
git tag  # 选择版本 git checkout 2.3.0d5 util/install.sh  # 执行安装
脚本
sudo mn  # 验证安装
```

1.2.2 SDN 控制器：RYU 控制器

Ryu 是一款开源 SDN 控制器，完全由 Python 语言实现，使用者可以用 Python 语言在其上实现自己的应用。Ryu 目前支持所有版本的 Openflow 协议。相较于其他的控制器，其优点为方便简洁，易于使用和开发。

下面介绍 ryu 控制器的安装工作

```
python-eventlet
python-routes
python-webob
python-paramiko  # 安装 RYU，需要安装一些 python 的套件

git clone git://github.com/osrg/ryu.git
cd ryu
sudo pip install -r tools/pip-requires
sudo python setup.py install  # 进行源码安装
cd  /ryu/ryu/app  # 进入目录
ryu-manager simple_switch.py  # 测试文件
```

第二章 方案设计

2.1 课程作业目的

本实验旨在通过对同一网络拓扑实现经典路由与基于 SDN 的路由，对比二者的性能差异，进而验证 SDN 相较于传统网络的灵活性与优越性。

2.2 场景设计与预期要求

2.2.1 实验场景设计

设计如下的小型网络拓扑进行实验：

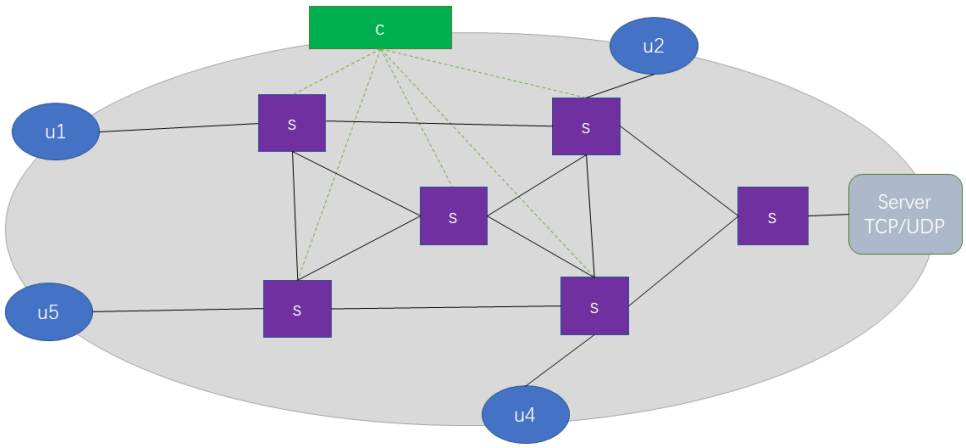

图 2-1 实验网络拓扑

其中 C 为控制器，S 为 SDN 交换机，u1-u5 为终端。我们设计以下两种路由策略：

1) 按目的地址路由：采用最短路径算法
2) 按多重约束路由：目的地址 + 传输层协议类型（UDP 通道，TCP 通道）

在测试时令 u1, u5 同时发送 TCP/UDP 流到服务器，分别在每个 SDN 交换机采集流量，分析传统 IP 路由和 SDN 路由的各项 QoS 性能指标。

2.2.2 实验预期要求

依照上述场景设计，实验应达到如下预期要求：

- 在 mininet 中依照实验场景搭建网络拓扑
- 在该拓扑上成功实现并运行按目的地址路由
- 在该拓扑上成功实现并运行按多重约束路由
- 实现一种通用的测量链路节点 QoS 性能指标的方法，以便对两种路由策略进行对比与评价

2.3 实现方案与技术细节

2.3.1 实现方案

2.3.1.1 拓扑设计

拓扑设计参照实验要求，实验时的拓扑如下：

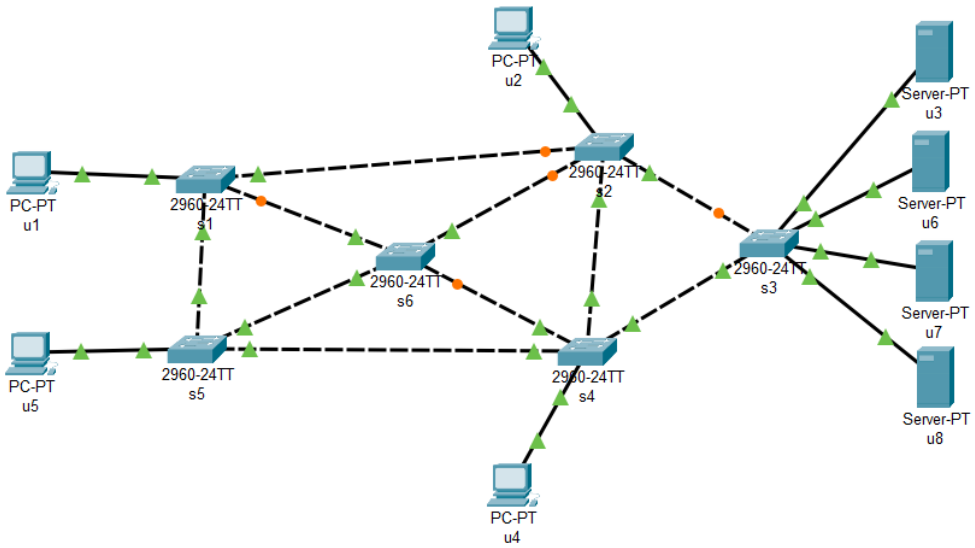

图 2-2 拓扑图

其中，u_3, u_6, u_7, u_8 作为 iperf3 测试时的服务器，u_1, u_5 作为 iperf3 测试时的客户端。

各链路均有带宽和延迟的参数，如下表所示：

	$s_1 - s_2$	$s_1 - s_5$	$s_2 - s_4$	$s_4 - s_6$	其他
延迟 (ms)	15	15	15	15	10
带宽 $(Mbps)$	50	50	50	50	75

表 2.1 延迟带宽表

2.3.1.2 传统路由方案设计

类似于[?]中所描述的进行路由的方法,使用 Ryu 控制器自带的 `ryu.app.rest_router`,可以进行传统的基于最短距离的路由。

`ryu.app.rest_router` 实现了两个功能:

- 生成路由规则并下发给 Switch。这样一来每个 Switch 都可以看作一个路由器。
- 支持通过一套 Restful API 对路由规则进行增删改查,这样一来外部可以根据需要修改路由规则。

但这个 APP 并不能自动地进行路由规则的生成,我们需要通过某种规则来生成路由并通过此 APP 的 Restful API 部署生成的路由规则。`classic/make_route.py` 文件就实现了根据最短路径算法生成路由并告知 `ryu.app.rest_router` 的过程。

主要的生成过程如下:

1) 把拓扑视作一个图,通过在每个点进行 Dijkstra 算法生成两点之间的最短路径。
2) 通过路径生成路由信息。如 u_1 到 u_3 的最短路径是 $u_1 \to u_2 \to u_3$,那么生成类似于 $(dst : u_3, next\_hop : u_2)$ 的路由信息。
3) 把生成的路由信息发送给 `ryu.app.rest_router`

这样的方法需要我们提前给定两个点之间的边权,这里把延迟作为边权。

2.3.1.3 基于 TCP 与 UDP 路由的 SDN 路由方案设计

首先给定一套把从 u_1 和 u_5 发出的 TCP 和 UDP 进行分流的方案:

如图,红色路径是 TCP 流的路径,蓝色路径是 UDP 流的路径。

实现这个路径需要用到 Ryu 控制器自带的 `ryu.app.ofctl_rest`,这个 APP 提供了一套增删改查任意 Switch 流表的 Restful API,使用它可以方便地创建所需的流表项。

在传统路由的基础上创建几个特定的高优先级流表项即可。

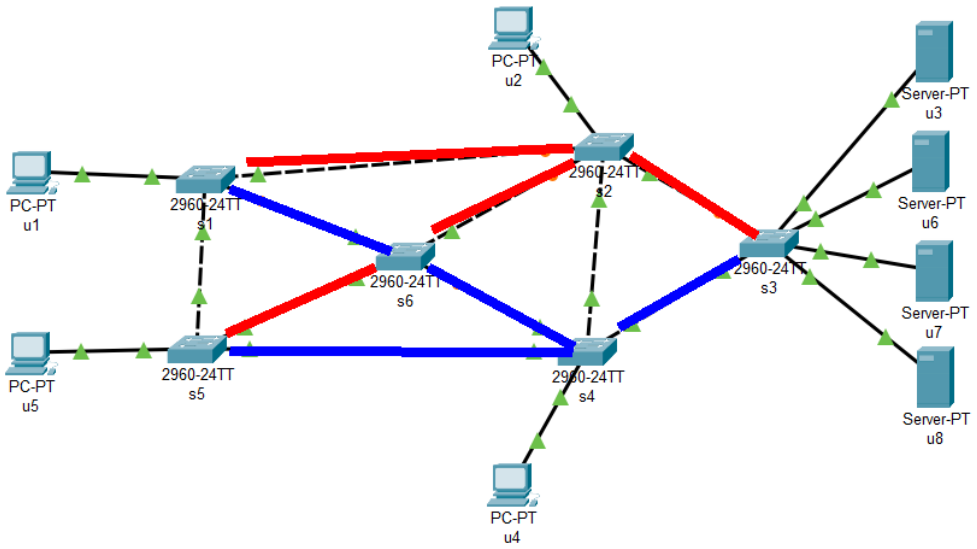

图 2-3 UDP 和 TCP 路径

2.3.1.4 Qos 性能测试设计

这里采用了 iperf3 工具进行测试。iperf3 工具可以创建一段时间的 TCP 和 UDP 通信，这一段通信的速率、延迟、丢包等信息可以帮助我们获取 Qos 性能信息。也可以在测试时实时进行报文抓取，这也反映了很多 Qos 性能信息。

2.3.2 实现传统路由

2.3.2.1 搭建拓扑

首先找到 `classic/create_topo.py` 这个文件，在配置好 mininet 的主机 A 中运行

```
sudo mn --custom create_topo.py --topo mytopo \
--switch ovs --controller remote,ip={ip of B} --link tc
```

其中 `ip` 项是控制器主机 B 的 ip 地址。

在 B 主机中运行

```
ryu run --observe-links  \
ryu.app.rest_router ryu.app.gui_topology.gui_topology
```

这样一来整个拓扑已经搭建成功。浏览 `http://ip_of_B:8080` 可以查看拓扑图像如下：

注意到这里没有画出主机的拓扑情况。

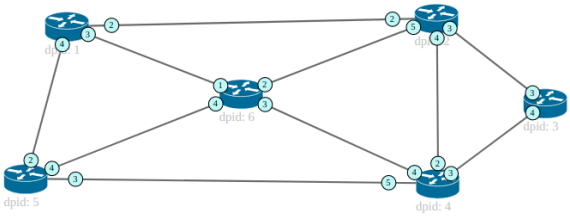

图 2-4 拓扑图

2.3.2.2 生成路由

首先找到 `classic/make_route.py`，在主机 B 上运行

```
python make_route.py
```

这会在控制器上创建基于最短路算法的静态路由。此时路由已经创建好，但是，由于主机的默认网关均未进行配置，要手动配置默认网关。

在 mininet 的命令行中运行

```
u1 ip route add default via 172.18.0.2 \
u2 ip route add default via 172.18.1.2 \
u3 ip route add default via 172.18.2.2 \
u4 ip route add default via 172.18.3.2 \
u5 ip route add default via 172.18.4.2 \
u6 ip route add default via 172.18.2.2 \
u7 ip route add default via 172.18.2.2 \
u8 ip route add default via 172.18.2.2
```

这样一来路由就搭建成功了。

2.3.2.3 测试路由

在 mininet 的命令行中运行

```
pingall
```

得到输出如下：

```
mininet> pingall
*** Ping: testing ping reachability
u1 -> u2 u3 u4 u5 u6 u7 u8
u2 -> u1 u3 u4 u5 u6 u7 u8
u3 -> u1 u2 u4 u5 u6 u7 u8
u4 -> u1 u2 u3 u5 u6 u7 u8
```

```
u5 -> u1 u2 u3 u4 u6 u7 u8
u6 -> u1 u2 u3 u4 u5 u7 u8
u7 -> u1 u2 u3 u4 u5 u6 u8
u8 -> u1 u2 u3 u4 u5 u6 u7
*** Results: 0% dropped (56/56 received)
```

这样一来就可以确定路由已经正常工作。

2.3.2.4 性能测试

在 mininet 的命令行中运行

```
u3 iperf3 -s -D
u6 iperf3 -s -D
u7 iperf3 -s -D
u8 iperf3 -s -D
u1 iperf3 -c u3 -t 20 > u1.tcp &
u1 iperf3 -c u6 -u -b 40M > u1.udp &
u2 iperf3 -c u7 -t 20 > u2.tcp &
u2 iperf3 -c u8 -u -b 40M > u2.udp &
```

性能测试的结果在 `u1.tcp, u1.udp, u2.tcp, u2.udp` 这四个文件中。

2.3.3 实现基于 UDP 和 TCP 选择的 SDN 路由

2.3.3.1 搭建拓扑

首先找到 `classic/create_topo.py` 这个文件，在配置好 mininet 的主机 A 中运行

```
sudo mn --custom create_topo.py --topo mytopo \
--switch ovs --controller remote,ip={ip of B} --link tc
```

其中 `ip` 项是控制器主机 B 的 ip 地址。

在 B 主机中运行

```
ryu run --observe-links  \
ryu.app.rest_router ryu.app.gui_topology.gui_topology \
ryu.app.ofctl_rest
```

这样一来整个拓扑已经搭建成功。实际上，这里的拓扑和传统路由时的拓扑没有区别。

2.3.3.2 生成路由

首先找到 `sdn-tcp-udp` 文件夹中的 `make_classic_route.py` 和 `make_sdn_route.py`，在主机 B 上运行

```
python make_classic_route.py \
python make_sdn_route.py
```

这会在控制器上创建基于最短路算法的静态路由和基于 UDP 和 TCP 的 SDN 路由。

此时路由已经创建好，但是，由于主机的默认网关均未进行配置，要手动配置默认网关。

在 mininet 的命令行中运行

```
u1 ip route add default via 172.18.0.2
u2 ip route add default via 172.18.1.2
u3 ip route add default via 172.18.2.2
u4 ip route add default via 172.18.3.2
u5 ip route add default via 172.18.4.2
u6 ip route add default via 172.18.2.2
u7 ip route add default via 172.18.2.2
u8 ip route add default via 172.18.2.2
```

这样一来路由就搭建成功了。

2.3.3.3 测试路由

在 mininet 的命令行中运行

```
pingall
```

得到输出如下：

```
mininet> pingall
*** Ping: testing ping reachability
u1 -> u2 u3 u4 u5 u6 u7 u8
u2 -> u1 u3 u4 u5 u6 u7 u8
u3 -> u1 u2 u4 u5 u6 u7 u8
u4 -> u1 u2 u3 u5 u6 u7 u8
```

```
u5 -> u1 u2 u3 u4 u6 u7 u8
u6 -> u1 u2 u3 u4 u5 u7 u8
u7 -> u1 u2 u3 u4 u5 u6 u8
u8 -> u1 u2 u3 u4 u5 u6 u7
*** Results: 0% dropped (56/56 received)
```

这样一来就可以确定路由已经正常工作。

2.3.3.4 性能测试

这里进行了两次性能测试, 一次测试和传统路由相同, 限制 udp 的速率为 $40Mbp/s$, 一次把 udp 的速率限制到 $100Mbp/s$。

在 mininet 的命令行中运行

```
u3 iperf3 -s -D
u6 iperf3 -s -D
u7 iperf3 -s -D
u8 iperf3 -s -D

u1 iperf3 -c u3 -w 1M -t 20 > u1-40.tcp &
u1 iperf3 -c u6 -u -b 40M > u1-40.udp &
u5 iperf3 -c u7 -w 1M -t 20 > u5-40.tcp &
u5 iperf3 -c u8 -u -b 40M > u5-40.udp &

u1 iperf3 -c u3 -w 1M -t 20 > u1-100.tcp &
u1 iperf3 -c u6 -u -b 100M > u1-100.udp &
u5 iperf3 -c u7 -w 1M -t 20 > u5-100.tcp &
u5 iperf3 -c u8 -u -b 100M > u5-100.udp &
```

第一次性能测试的结果在 `u1-40.tcp, u1-40.udp, u5-40.tcp, u5-40.udp` 这四个文件中。

第二次性能测试的结果在 `u1-100.tcp, u1-100.udp, u5-100.tcp, u5-100.udp` 这四个文件中。

2.4 实验中遇到的挑战与问题及其解决方法

2.4.1 Ryu 控制器的内部问题

在金钰同学的计算机上测试时，Ryu 控制器的内置 `ryu.app.rest_router` 不能正常地进行工作。报错信息为 python 中的 `KeyError`，即一个哈希表的键不存在。这个问题并不源于 `ryu.app.rest_router`，真正的错误来源是 Ryu 控制器的内部组件。这个错误也在使用半自动测试脚本时出现了。

在李晨曦同学的计算机上测试时，Ryu 控制器的内置 `ryu.app.gui_topology` 不能正常地工作。表现为浏览 `http://ip_of_B:8080` 后，应该出现拓扑图的区域空无一物。

这两个问题都涉及到 Ryu 控制器的内部问题。李晨曦同学通过重新安装了 Ryu 控制器解决了问题，金钰同学通过拷贝李晨曦同学主机上的的 Ryu 控制器并覆盖到相应位置解决了问题。

最终，使用半自动测试脚本时的问题通过在启动 Mininet 之后再启动 `ryu.app.rest_router` 得到了解决。

2.4.2 Mininet 内置 Shell 的不便

Mininet 内置的 Shell 语法类似于

$$host\ command$$

乍看之下，这没有什么问题。然而，实际上这有比较严重的问题。首先，这样的设计使得它无法正常地使用 Shell 脚本批量地对**不同主机**执行命令。这导致每次配置拓扑中主机的默认网关时都要手动输入 8 条命令。

其次，无论是 `&&` 操作符还是 \ 分行符号，在 Mininte Shell 中的语义均是『在一个主机上执行』。也就是说，

```
u1 ls . && u2 cd ..
```

会被理解为在 u_1 上执行

```
ls . && u2 cd ..
```

这样导致无法通过『复制粘贴写好的命令』来解决这个问题。

为了解决这个问题，我们用 python 写了一个半自动测试脚本 `in.py`，它可以完

成启动 Mininet、配置默认路由、进行测试等等一系列需要使用 Mininet 命令行的操作。然而，这个脚本并不是全自动的。这是因为每次测试时都要启动控制器，而控制器和 Mininet 并不在一个主机上，这个脚本再 Mininet 的主机上运行，想要启动控制器是不太方便的。

当然，这也可以通过在部署控制器的主机上运行一个服务器程序解决，但是我们最终没有采用这个方案，因为在半自动测试脚本的帮助下，测试过程已经足够简单了。

第三章 实验分析

3.1 跳数

3.1.1 传统路由的情况

由于在传统路由中，路由条数是完全确定的，且由于这里使用的图是无向图，因而路由是完全对称的。这样一来我们只需要分析 u_1 到 u_3 的路径和 u_5 到 u_6 的路径，即可确定跳数。

经过对图的分析，知路径和跳数如下：

	跳数	路径
$u_1 - u_3$	3	$u_1 \to s_1 \to s_2 \to s_3 \to u_3$
$u_5 - u_6$	3	$u_1 \to s_5 \to s_4 \to s_3 \to u_6$

表 3.1 传统路由情况下的条数路径表

3.1.2 SDN 路由的情况

在 SDN 路由中，我们更改了 TCP 和 UDP 的路由。故此处要分析四种路径：

	跳数	路径
$u_1 - u_3(\mathsf{TCP})$	3	$u_1 \to s_1 \to s_2 \to s_3 \to u_3$
$u_1 - u_3(\mathsf{UDP})$	4	$u_1 \to s_1 \to s_6 \to s_4 \to s_3 \to u_3$
$u_5 - u_6(\mathsf{TCP})$	4	$u_1 \to s_5 \to s_6 \to s_2 \to s_3 \to u_6$
$u_5 - u_6(\mathsf{UDP})$	3	$u_1 \to s_5 \to s_4 \to s_3 \to u_6$

表 3.2 SDN 路由情况下的条数路径表

3.1.3 对比

从上面的两个表格可以很清楚地看到，使用 SDN 反而会使得某些情况下的条数变多。但是从另外的角度上来说，如果不使用 SDN，那么实际上只使用了一部分线路，没有充分利用线路资源。

3.2 带宽

3.2.1 总体情况

由于 Mininet 是网络模拟器而非真实网络，故此处的带宽是我们自定义的带宽。实际上，这个带宽仅对 TCP 流量有效，对 UDP 流量完全没有效果。我们曾经用 $1Gbps$（前面已经叙述过，链路带宽上限为 $75Mbps$）的 UDP 流量进行测试，结果仍为能够全速传输，且用 Wireshark 抓包分析后确实产生了相当的流量。故对于 UDP 流量的带宽分析是没有意义的。

另外一个问题是，这里有发送带宽和接收带宽，以何者为准呢？理论上应以接收带宽为准，然而首先发送带宽和接受带宽差别并不大，其次 iperf3 只能实时测得发送带宽，故这里以发送带宽为准。

3.2.2 传统路由的情况

在传统路由时，我们同时让 $u1$ 和 $u2$ 进行 UDP 和 TCP 测试，带宽-时间图为图 3-1 和图 3-2。

当然，为了证明我们对于 UDP 流量的分析所言不虚，图 3-3 展示了 UDP 流量的带宽-时间图。

从图中可以看到，TCP 流量测试持续时间为 20 秒，UDP 流量测试持续时间为 10 秒。它们同时进行，这样一来在前 10 秒内 TCP 流量被 UDP 挤占，基本不传输内容。在后 10 秒内，带宽在 $8-14Mbps$ 左右波动，进行正常传输。

平均带宽如下表：

测试	带宽
u_1(TCP)	6.44
u_5(TCP)	6.48

表 3.3 传统路由情况下的条数路径表

3.2.3 SDN 路由的情况

在 SDN 路由时，我们进行了四次测试，分别让 UDP 发送的带宽为 $40Mbps$ 和 $100Mbps$，这时 $u1$、$u5$ 的 TCP 带宽-时间图分别为图 3-4、图 3-5、图 3-6 和图 3-7。

可以看到，虽然带宽有一定波动，但是总体来看还是较为平稳的。

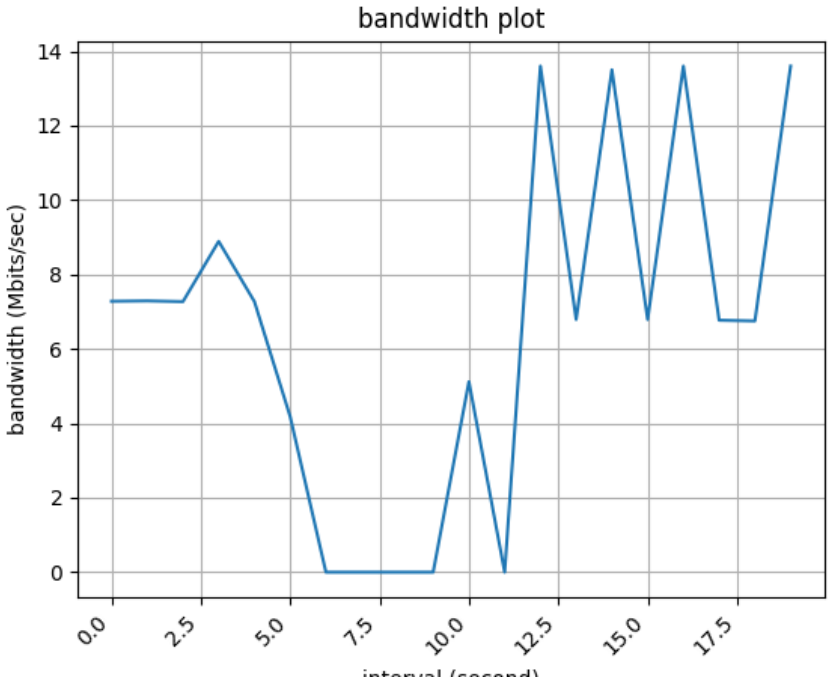

图 3-1 u_1 进行 tcp 流量测试带宽-时间图

平均带宽如下表:

测试	带宽
u_1(`TCP,UDP=40`)	28.1
u_1(`TCP,UDP=100`)	33.8
u_5(`TCP,UDP=40`)	26.6
u_5(`TCP,UDP=100`)	29.2

表 3.4 SDN 路由情况下的平均带宽表

3.2.4 对比

从传统路由的实验中，我们可以清楚地看到 UDP 流量对 TCP 流量的干扰是非常严重的。但在 SDN 路由实验中，由于我们将 TCP 和 UDP 进行了分流，这样的干扰不复存在。这可以从两个方面得到证实:

1) 在 SDN 路由中，增加 UDP 的带宽，平均带宽没有受到任何影响
2) 在 SDN 路由中，平均带宽在前 10 秒（UDP 存在）和后 10 秒（UDP 不存在）没有太大差别。

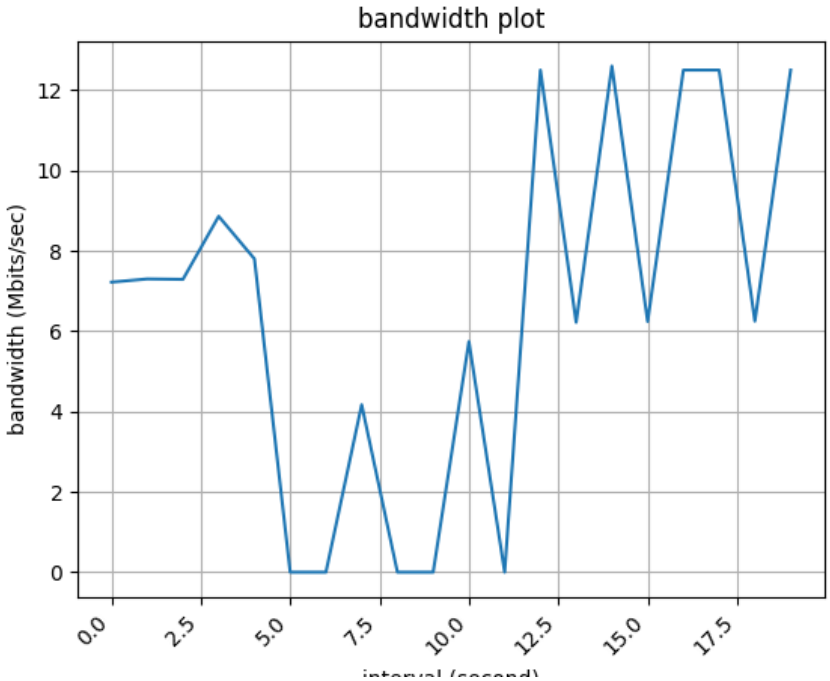

图 3-2 u_5 进行 tcp 流量测试带宽-时间图

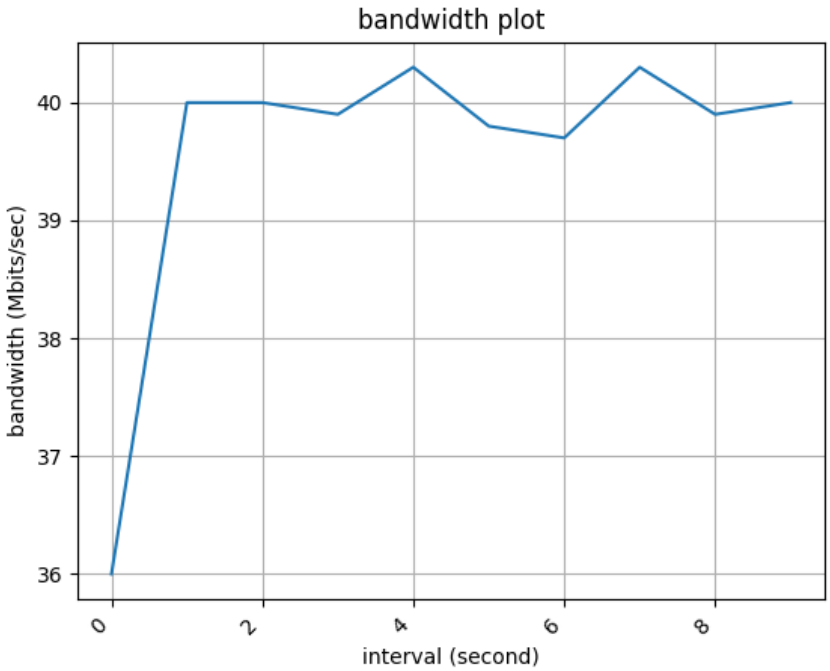

图 3-3 u_1 进行 udp 流量测试带宽-时间图

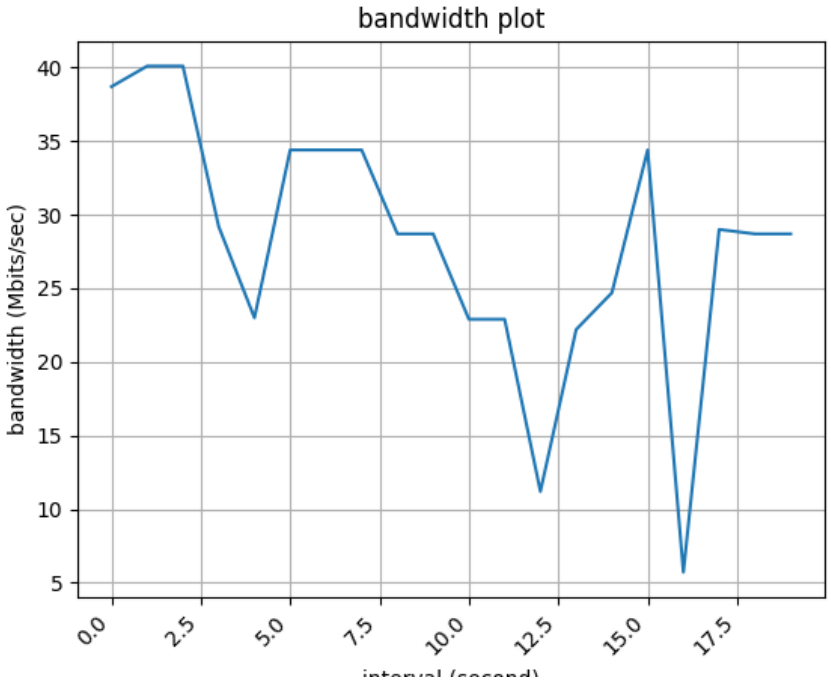

图 3-4 u_1 在 udp 带宽为 $40Mbps$ 时进行 tcp 流量测试带宽-时间图

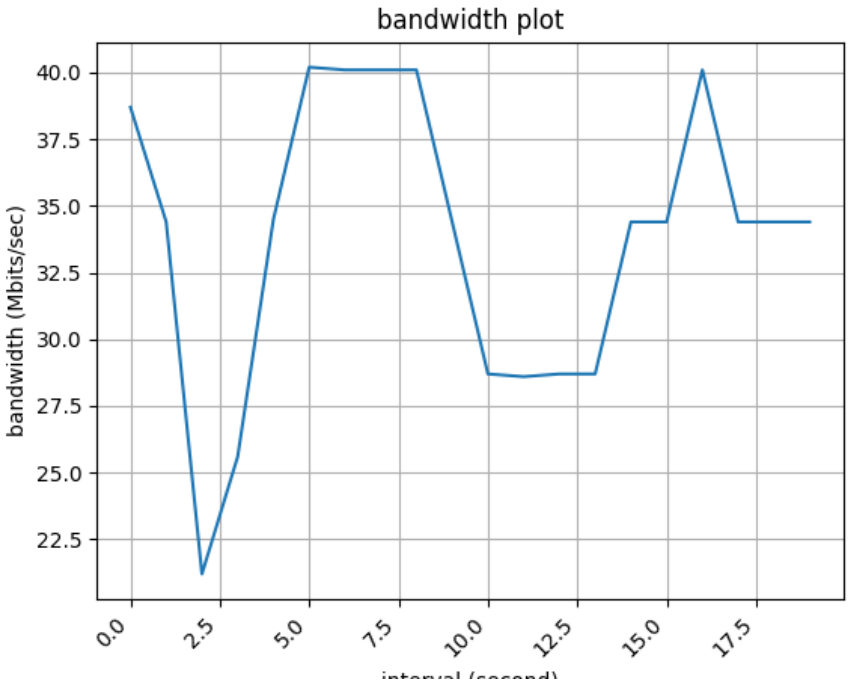

图 3-5 u_1 在 udp 带宽为 $100Mbps$ 时进行 tcp 流量测试带宽-时间图

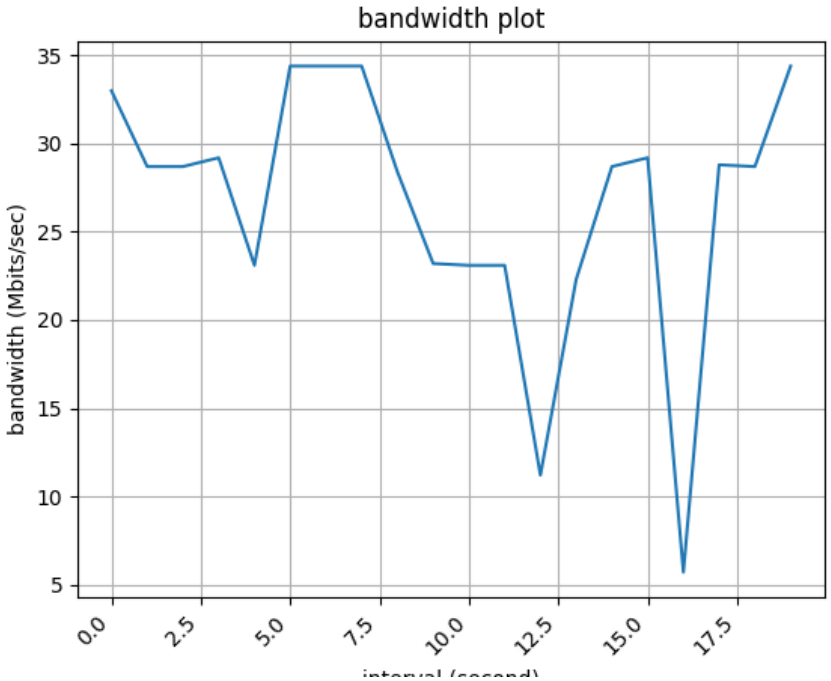

图 3-6 u_5 在 udp 带宽为 $40Mbps$ 时进行 tcp 流量测试带宽-时间图

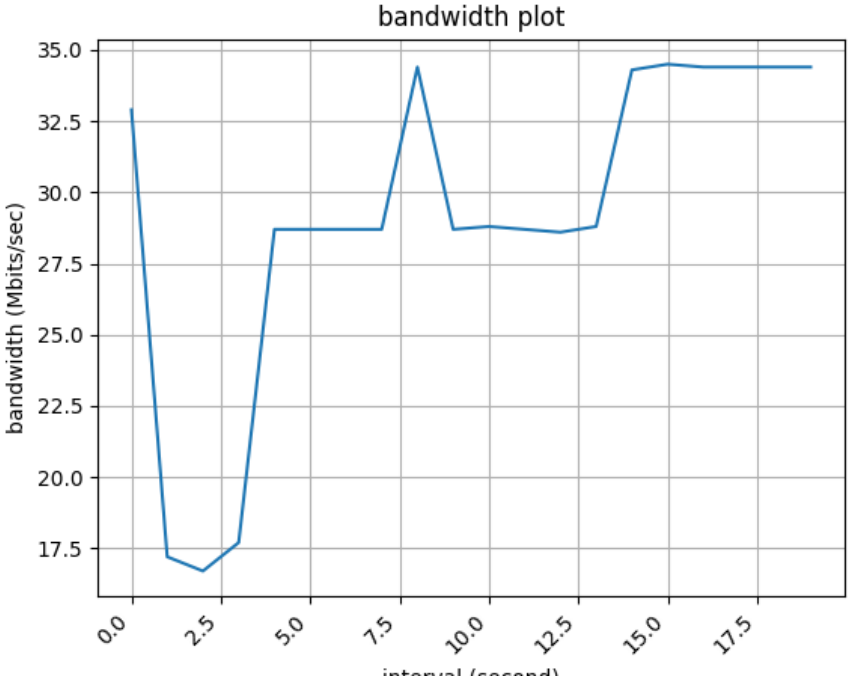

图 3-7 u_5 在 udp 带宽为 $100Mbps$ 时进行 tcp 流量测试带宽-时间图

3.3 延迟

延迟和跳数一样，理论上都是完全确定的。因为每条路径的延迟是通过 Mininet 的启动脚本进行设置的，而 Switch 的延迟可以忽略不计，故是完全确定的。然而测试后发现延迟并不是完全确定的，这可能是由于 Switch 的延迟不是可忽略的，或 Mininet 内部有其他机制。

我们进行了两种延迟测试，由于 SDN 路由时并不对 ICMP 报文进行特殊处理，这里只能观察到传统路由下的延迟。

$u1$ 到 $u2$ 的延迟理论上为 15 ms，实际上的延迟测试结果如下：

```
mininet> u1 ping u2
PING 172.18.1.1 (172.18.1.1) 56(84) bytes of data.
64 bytes from 172.18.1.1: icmp_seq=1 ttl=64 time=40.0 ms
64 bytes from 172.18.1.1: icmp_seq=2 ttl=64 time=35.0 ms
64 bytes from 172.18.1.1: icmp_seq=3 ttl=64 time=30.9 ms
64 bytes from 172.18.1.1: icmp_seq=4 ttl=64 time=30.7 ms
64 bytes from 172.18.1.1: icmp_seq=5 ttl=64 time=38.3 ms
64 bytes from 172.18.1.1: icmp_seq=6 ttl=64 time=34.8 ms
64 bytes from 172.18.1.1: icmp_seq=7 ttl=64 time=30.3 ms
```

图 3-8 $u1$ ping $u2$ 的结果

由于 ping 的时间是往返时间，所以这里的理论值为 30 ms. 可以看到，这和理论值有一定出入。

3.4 丢包

丢包实际上有两种情况：

- 线路丢包或由于缓冲区满造成的丢包
- 由于 TCP 的特性，某些包虽然被接受方的网络层接收，但没有被传输层接收（比如说序列号在当前接收窗口之前的报文）

这两种情况都可以用 η 度量，η 的计算公式为：

$$\eta = \frac{D_{recv}}{D_{send}}$$

其中 D_{recv} 是接收方收到的数据，D_{send} 是发送方发送的数据。

显然地，η 越接近于 1，丢包率就越低。这里计算出 TCP 传输时的 η 如表 3.5.

可见，SDN 路由由于对 TCP 和 UDP 进行了分流，能够有效地解决 UDP 对 TCP 有干扰造成的丢包问题，提高 η 值。

测试情况	测试对象	η
传统路由，$\mathsf{UDP} = 40Mbps$	$u_1 \to u_3$	0.882
传统路由，$\mathsf{UDP} = 40Mbps$	$u_5 \to u_7$	0.903
SDN 路由，$\mathsf{UDP} = 40Mbps$	$u_1 \to u_3$	0.997
SDN 路由，$\mathsf{UDP} = 40Mbps$	$u_5 \to u_7$	0.987
SDN 路由，$\mathsf{UDP} = 100Mbps$	$u_1 \to u_3$	0.991
SDN 路由，$\mathsf{UDP} = 100Mbps$	$u_5 \to u_7$	0.991

表 3.5 η 数值表

第四章 总结

本次实验难度较大且复杂。首先，本次实验所需要掌握的软件与工程原理较多，需要进行许多提前学习；其次，由于整个实验在 Ubuntu 环境下进行运行且代码版本多样，本次实验的运行环境并不完善，造成我们花费了大量时间进行工程性的调试，进行环境配置等工作；最后，本次实验中我们所需要熟练应用的控制器需要我们阅读相关的文档，实现基于 python 的控制器编程。这次的实验，我们小组比上一次讨论更加多，一同协作来完成任务，收获颇丰。

完成实验并进行相关分析后，我们对 SDN 网络控制器的配置，mininet 的用法掌握得更加熟练。我们更好地理解了 SDN 相较于传统路由所带来的重大突破，虽然我们现在仅仅使用了非常简单的拓扑来复现整个流程，但是这个依旧具有很好的参考价值。在跳数方面，SDN 会相较于传统路由让数据包经过更多的跳数，这是因为 SDN 尝试着均衡网络资源而造成的。带宽方面，在仿真的环境中带宽可以任意设置，但是在实际的应用中带宽总是有限的，均衡网络资源能够让所有的网络带宽得到充分的利用，有效避免了拥塞情况的发生。所以 SDN 的带宽会更加平稳，传统路由中 TCP 被 UDP 流量冲击的现象也不复产生。延迟和丢包同理，SDN 路由都会因为其进行了合理的调配而得到改善。

本次实验就此完成。感谢小组成员之间的互相帮助，互相配合。也非常感谢老师在这两个学期的教学与对我们问题耐心的解释。计算机网络是我们学习生活中的重要课程，在未来许多时候都会重新需要使用。希望自己能够时时温故知新。现在，我们需要去进行期末考试的备考了，希望自己能考出一个好成绩