

第三题：指定业务带宽保障

目录：

第一章 背景介绍.....	2
1.1 实验背景.....	2
1.2 实验目的.....	2
1.3 实验环境搭建.....	2
第二章 可行性分析.....	4
2.1 Floodlight 对 Openvswitch 的控制.....	4
2.2 OpenvSwitch 对 QoS 策略的支持.....	4
第三章 实验方案设计.....	5
3.1 OpenFlow QoS 总体方案设计.....	5
3.2 控制平面功能设计	7
3.3 转发平面功能设计	7
第四章 OpenFlow QoS 实现.....	10
4.1 QoS 控制器模块实现.....	10
4.2 CLI 指令配置模块实现.....	11
4.3 DiffServ 流量控制模块实现.....	12
第五章 OpenFlow QoS 功能测试	15
5.1 系统测试环境介绍.....	15
5.1.1 测试平台.....	15
5.1.2 实验拓扑.....	15
5.2 实验测试方法.....	16
5.2.1 网络流量测试工具.....	16
5.2.2 流量控制功能验证方法.....	17
5.3 流量控制功能验证.....	17
5.3.1 系统端口速率 TCP 限速测试.....	17
5.3.2 系统端口 TCP 带宽保障测试.....	20
5.3.3 系统视频流速率带宽保障测试.....	22
5.4 实验总结.....	23
附录 流量种类 Map 表.....	25

第一章：背景介绍

1.1、实验背景

数据中心提供多种业务，但一般只进行尽力而为的转发，不单独为某一业务带宽提供额外的保障，这就造成某些关键性业务无法得到很好地保证（如视频业务），可能影响业务的正常运转（视频不流畅）。

近年来，随着网络技术的快速发展，网络业务种类日益多元化，网络带宽容量需求日益扩大。某些国内最大的海量视频数据流媒体，每天 2-2.5 亿视频播放量，千万级用户访问行为和视频播放需求，为了能够更好地满足用户对于视频流的播放需求，基于互联网架构的流媒体服务应该具有更好的 QoS 服务保证。新型网络业务的广泛应用，包括视频通话、电话会议、VOIP 等，使得人们对服务质量（Quality of Service, QoS）要求越来越高。网络如何有效保障业务的服务质量受到越来越多人的关注。随着新型业务的多元化，传统网络体系架构暴露出了越来越多的弊病，例如网络结构复杂化和网络设备性能的饱和，对网络新业务的推广提出了严峻的考验，已经不能满足各种新型业务对服务质量的需求。

因此，一场新型网络体系架构的革命蓄势待发，软件自定义网络——一种可编程网络体系架构应运而生。OpenFlow 是 SDN 的产物，是一种新型网络交换模型，该模型利用开放流表（FlowTable）实现用户对网络数据处理的可编程控制。针对当前 OpenFlow 网络对 QoS 管理的需求，结合对传统 IP 网络区分服务模式（DiffServ）QoS 技术的研究，设计并实现了一套基于 DiffServ 模型和 OpenFlow 网络架构的 QoS 系统。按照预定的设计方案通过搭建小型网络拓扑对其进行验证，并对实验结果进行分析，验证设计的 QoS 系统显著的提高了对网络 QoS 性能的全局掌控，降低了底层交换设备的工作负载。

1.2、实验目的

带宽保障属于 QoS 的一种，本实验包含多种 QoS 策略。下文统一称为 QoS。对数据中心中提供的某种业务（如视频业务）进行带宽预留与保障，当总体流量大于链路承载能力时，优先保证指定业务的带宽。

针对视频传输控制，实时传输协议（RTP）、实时传输控制协议（RTCP）、实时流协议（RTSP）、资源预订协议（RSVP）等协议是基于 TCP 协议，且现在越来越多的视频流基于 TCP 协议传输。

本实验将完成 TCP 流量带宽限制、带宽保障和特定 Video Packet（视频流）流量的保障。

1.3、实验环境搭建

为了验证网络流量控制的性能，搭建了一个简单的 DiffServ 的小型网络，如图所示：

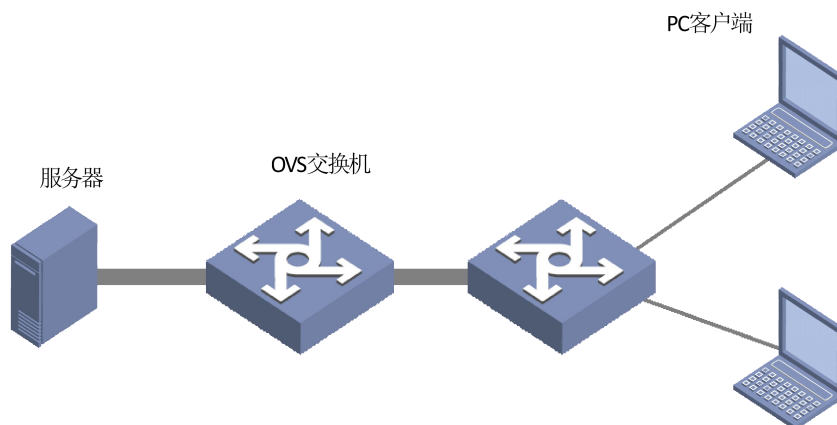


图 1.1 实验拓扑图

其中 OpenFlow 控制器为运行 Floodlight 控制器程序的 Linux(Ubuntu)主机，Floodlight 和 OVS 为运行 OpenFlow 网络的控制器和交换机。OVS 与控制器直连，提供多种服务的服服务集群，PC1 和 PC2 分别连接在右 OVS 上。实验结果通过数据流获得的网络带宽来验证和转发的速率来验证。从服务器向 PC1 发送 2 种优先级不同的 TCP 流 A、B，网络总带宽为 10Gbit/s。在 PC1 上通过对 A、B 流量的分析，得到实验数据。

从实验结果可知，QoS 系统能够正确地区分不同的 QoS 等级的服务，并提供差别服务质量保证。QoS 使用前，不同的流公平占用网络带宽，QoS 使用后，不同的 QoS 等级的数据流获得的网络带宽不同。从而实现对数据中心中提供的某种业务（如视频业务）进行带宽预留与保障。

第二章：可行性分析

2.1 Floodlight 对 Openvswitch 的控制

OpenFlow 协议支持控制器到交换机消息交互，controller-to-switch 消息由控制器发起，用来管理或获取交换机的状态，主要包括 6 种其子消息类型。Features 在建立传输层安全会话（Transport Layer Security Session）的时候，控制器发送 feature 请求消息给交换机，交换机需要应答自身支持的功能。

（一）Configuration 控制器设置或查询交换机上的配置信息。交换机仅需要应答查询消息。

（二）Modify-state 控制器管理交换机流表项和端口状态等。

（三）Read-state 控制器向交换机请求一些诸如流、网包等统计信息。

（四）Packet-out 控制器通过交换机指定端口发出网包。

（五）Barrier 控制器确保消息依赖满足，或接收完成操作的通知。

（六）asynchronous 消息由 switch 发起，用来将网络事件或交换机状态变化更新到控制器。

交换机与控制器通过被动或主动的控制进行状态信息更新、流表更新，是得控制器对交换设备进行网络拓扑管理、路由控制等，实现控制端对全局网络的实时掌控。

2.2 OpenvSwitch 对 QoS 策略的支持

QoS 中的流量监管（Traffic Policing）就是对流量进行控制，通过监督进入网络端口的流量速率，对超出部分的流量进行“惩罚”（这个惩罚可以是丢弃、也可是延迟发送），使进入端口的流量被限制在一个合理的范围之内。例如可以限制 TCP 报文不能占用超过 50% 的网络带宽，否则 QoS 流量监管功能可以选择丢弃报文，或重新配置报文的优先级。

QoS 流量监管功能是采用令牌桶（Token-Bucket）机制进行的。这里的“令牌桶”是指 OpenvSwitch 的内部存储池，而“令牌”则是指以给定速率填充令牌桶的虚拟信息包。

交换机在接收每个帧时都将添加一个令牌到令牌桶中，但这个令牌桶底部有一个孔，不断地按你指定作为平均通信速率（单位为 b/s）的速度领出令牌（也就是从桶中删除令牌的意思）。在每次向令牌桶中添加新的令牌包时，交换机都会检查令牌桶中是否有足够容量，如果没有足够的空间，包将被标记为不符合规定的包，这时在包上将发生指定监管器中规定的行为（丢弃或标记）。

在 OpenvSwitch 中采用 HTB，Hierarchical Token Bucket 令牌桶机制来保障和限制流量的带宽。后文将详细说明 HTB 机制在 Queue 队列上的应用。

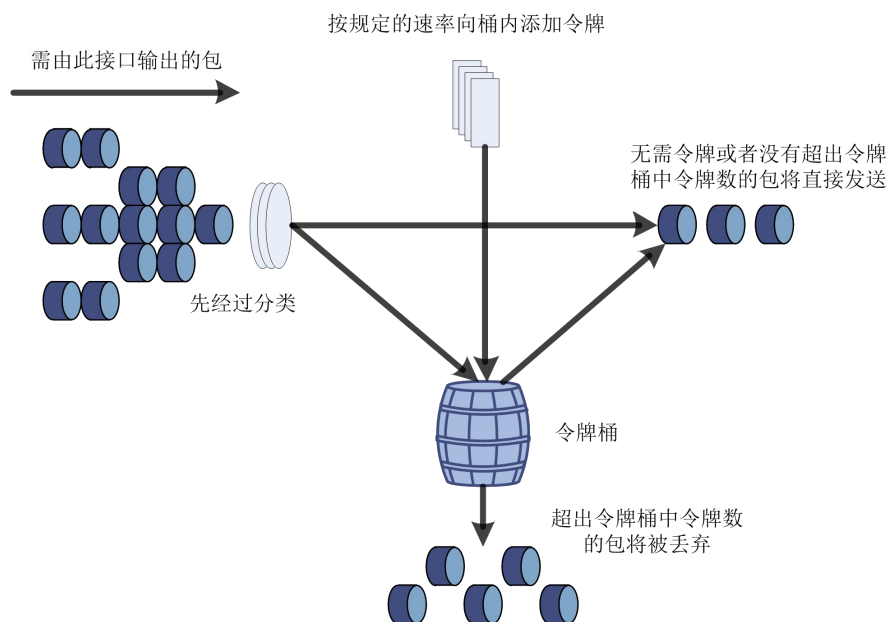


图 2.1 TB 的基本工作原理

第三章：实验方案设计

3.1 OpenFlow QoS 总体方案设计

DiffServ 在实现上由 PHB、包的分类机制和流量控制功能三个功能模块组成，其中流量控制功能包括测量、标记、整形和策略控制。当数据流进入 DiffServ 网络时，OpenvSwitch 通过标识 IP 数据包报头的服务编码点（Type of Service, ToS）将 IP 包划分为不同的服务类别，作为业务类别分类的标示符。

当网络中的其他 OpenvSwitch 在收到该 IP 包时，则根据该字段所标识的服务类别将其放入不同的队列，并由作用于输出队列的流量管理机制按事先设定的带宽、缓冲处理控制每个队列，即给予不同的每一跳行为（Per-Hop Behavior, PHB）。

在实际应用时，DiffServ 将 IPv4 协议中 IP 服务类型字段（TOS），作为业务类别分类的标示符。理论上，用户可以在 0x000000 至 0xffffffff 范围内为每个区分服务编码点对应的服务级别分配任意 PHB 行为。每个服务等级为分类的业务流提供不同的 QoS 保证，如下所示：

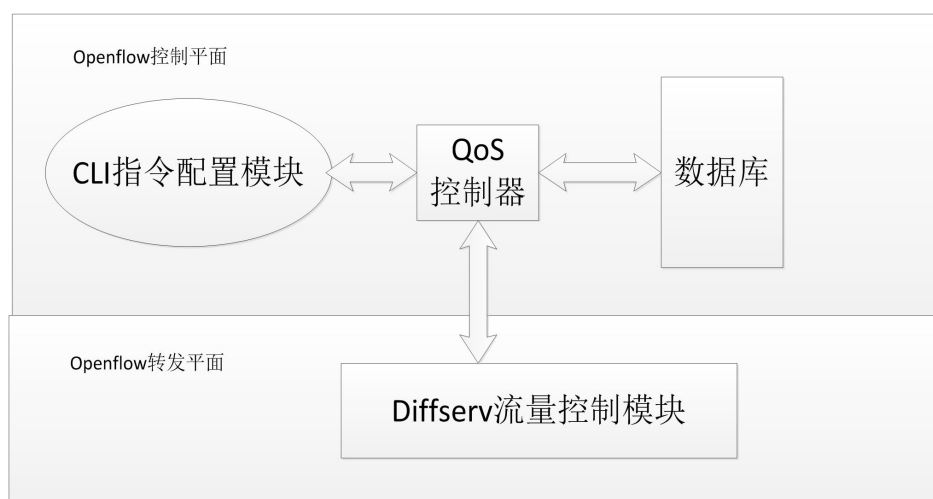


图 3.1 Openflow QoS 平面结构图

在实际应用时，具体工作流程如下：

(1) OVS 对业务进行转发，同时运行在入端口上的策略单元，对接受到的业务流进行测量和监控，查询数据业务是否遵循了 SLA，并依据测量的结果对业务流进行整形、丢弃和重新标记等工作。这一过程称为流量调整（traffic conditioning, TC）或流量策略（traffic policing, TP）。

(2) 业务流在入端口进行了流量调整后，再对其 DSCP 字段进行检查，根据检查结果与本地 SLA 等级条约进行对比并选择特定的 PHB。根据 PHB，所指定的排队策略，将不同服务等级的业务流送入 OpenvSwitch 出端口上的不同输出队列进行排队处理，并遵循约定好带宽缓存及调度。当网络发生拥塞时，还需要按照 PHB 对应的丢弃策略为不同等级的数据包提供差别的丢弃操作。

(3) 当业务流进入到 OpenvSwitch 时，只需根据 DSCP 字段进行业务分类，并选择特定 PHB，获得指定的流量调整、队列调度和丢弃操作。最后业务流进入网络中的下一跳，获得类似的 DiffServ 处理。

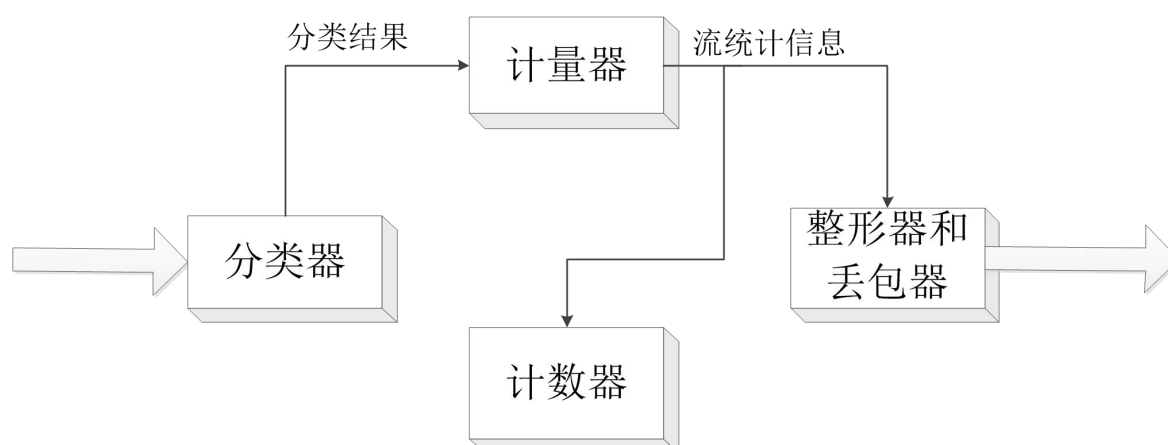


图 3.2 数据包分类和调节示意图

3.2 控制平面功能设计

QoS 控制负责对接收到指令进行解析和执行，实现集中式控制、布式处理。交换设备端口处流量控制分为入端口处流量整形和出端口处队列管理和调度。其中，交换机入端口处的流量整形和限制，具体通过调用 `ovsdb` 对应接口函数对入端口速率限制和入端口突发量等参数进行设置来实现。交换机出端口处主要采用队列管理和队列调度机制对流出网络的数据流进行控制。

队列管理和队列调度是流量调度的两个关键环节，为了在分配有限的网络资源时，保证业务流之间的公平性。队列管理为入队的报文分配缓存，当缓存溢出时对报文进行丢弃，目的是减小因排队造成的端点间的延迟。队列调度负责通过预设的调度算法将队列中等待处理的分组调度到相应的输出链路上。

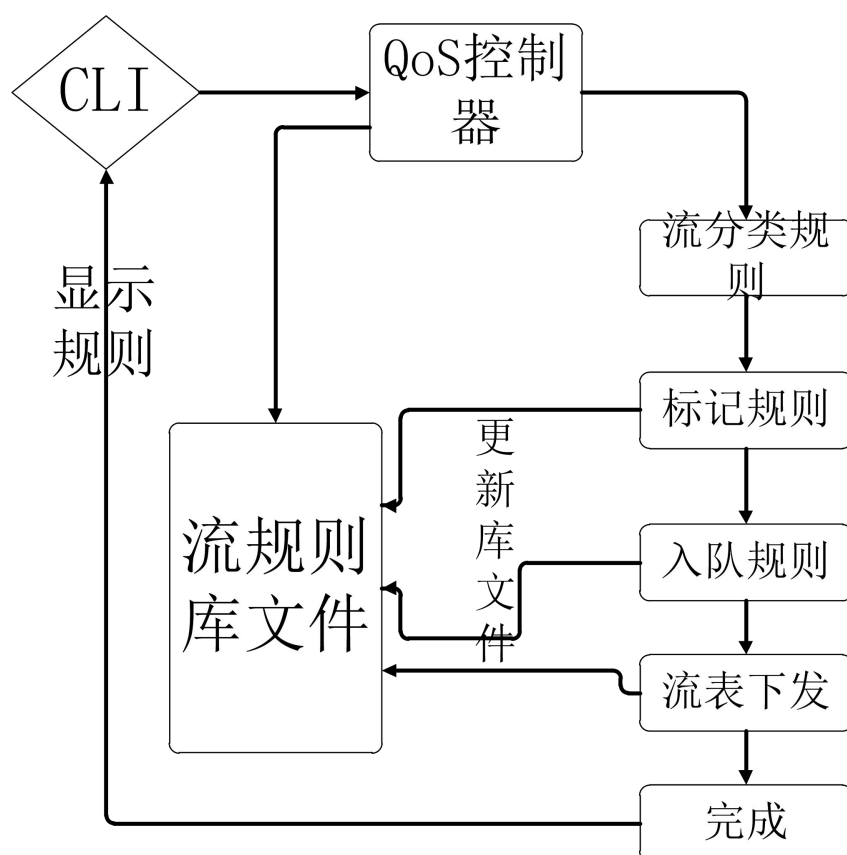


图 3.3 QoS 控制平面流程图

3.3 转发平面功能设计

Linux 内核流量机制提供了多种队列管理和队列调度的算法。其中无类的队列规则能够实现队列管理功能，如 RED、WRED 算法，分类的队列规则能够实现队列调度功能，如 HTB、CBQ、PRIO 算法。本系统中交换机出端口处的队列调度和队列管理机制的实现，我们采用了 HTB（Hierarchical Token Bucket）队列调度算法，实现底层转发结点上的队列管理。其中，队列调度机制通过对 Open vSwitch 的 `linux-htb` 队列模块进行配置来实现。

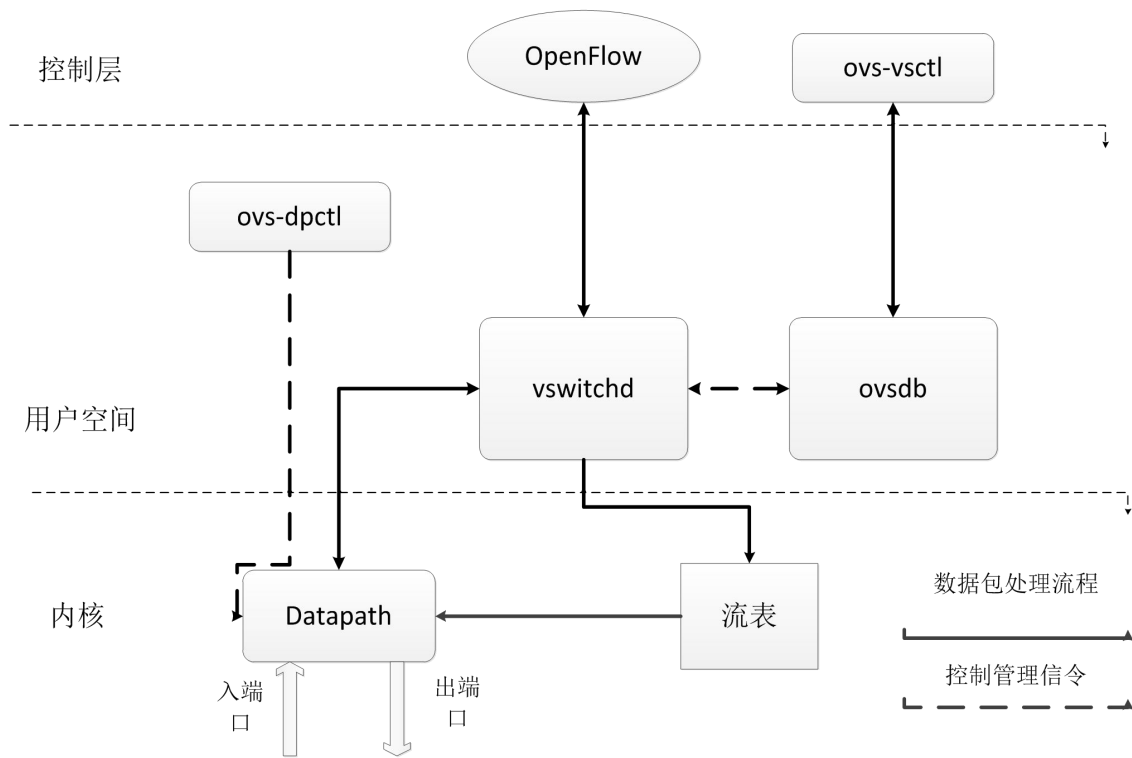


图 3.4 OpenvSwitch 交换机转发流程图

交换机端口处的流量整形和限制，队列调度机制通过对 OpenvSwitch 的 **linux-htb** 队列模块进行配置来实现。

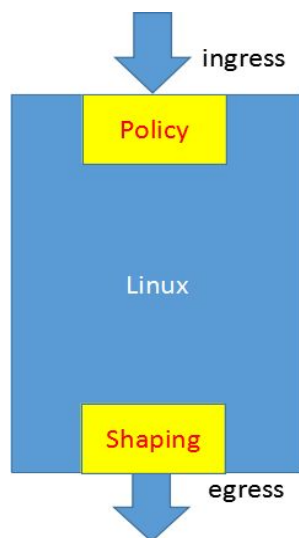


图 3.5 OpenvSwitch 交换机对流量整形

DiffServ 模型中对入端口处的流量整形和丢弃，出端口处的流量调度和丢弃处理分别以流量整形模块、队列管理模块和队列调度模块进行部署。这几个模块对业务流进行直接的操作，前提是依据控制的分类、标记和入队操作。其中，流量整形模块利用入端口处的流量策略控制机制实现，出端口处的队列管理模块采用分层令牌桶 (Hierarchical Token Bucket, HTB) 调度算法进行实现。这些组件 OpenFlow 交换机上进行实现。

核心策略 HTB, Hierarchical Token Bucket 功能组件：

- (一) Shaping: 仅仅发生在叶子节点，依赖于其他的 Queue
- (二) Borrowing: 当网络资源空闲的时候，借点过来为我所用
- (三) Rate: 设定的发送速度
- (四) Ceil: 最大的速度，和 rate 之间的差是最多能向其他流量借多少

Hierarchical Token Bucket (HTB)

Class structure and Borrowing

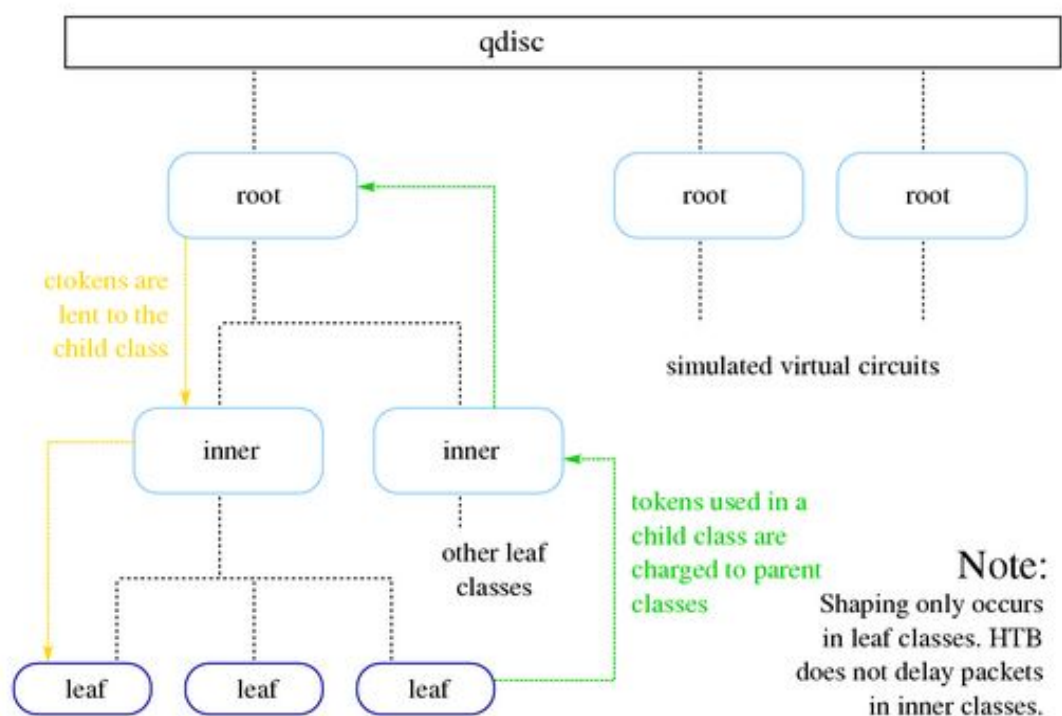


图 3.6 OpenvSwitch 交换机 HTB 实现

在 CLI 上输入 Queue 队列命令如下：

```
sudo ovs-vsctl -- set port s1-eth1 qos=@defaultqos -- set port s1-eth2 qos=@defaultqos -- --id=@defaultqos create qos type=linux-htb
```

```
other-config:max-rate=1000000000 queues=0=@q0,1=@q1,2=@q2 -- --id=@q0
create queue other-config:min-rate=1000000000 other-config:max-rate=1000000000
-- --id=@q1 create queue other-config:max-rate=200000000 -- --id=@q2 create queue
other-config:max-rate=2000000 other-config:mix-rate=200000
```

分别在 OpenvSwitch 的端口出创建三个 Queue 队列机制，速率则可以由自己定义。

第四章：OpenFlow QoS 实现

4.1 QoS 控制器模块实现

在整个管理系统中 QoS 控制管理模块处于核心的地位，部署在 OpenFlow 控制器上，提供 DiffServ 模型的流表控制管理功能。OpenFlow 网络的流表下发等控制行为是通过 QoS 控制器来决策并完成的。其中，流表的下发需要控制器和被控制的 OpenFlow 交换机通过 OpenFlow 协议规范来完成。

本系统设计并实现的基于 DiffServ 模型的流量控制模块提供以下基本的流量控制功能：

- (1) 限制特定流量带宽和速率；
- (2) 保障特定用户、服务或客户端的带宽；
- (3) 保障特定视频流的带宽；

QoS 控制器的 QoSPolicy 代码：

```
Map<String, Object> row;

        IResultSet policySet = storageSource
                .executeQuery(TABLE_NAME, ColumnNames, null,
null );
        /*从 strogeSource 中读信息 */
        for( Iterator<IResultSet> iter = policySet.iterator(); iter.hasNext();){
            row = iter.next().getRow();/*遍历信息*/
```

```

        QoSPolicy p = new QoSPolicy();
        if(!row.containsKey(COLUMN_POLID)
            || !row.containsKey(COLUMN_SW)/* 获取 OVS 的
ID*/
            || !row.containsKey(COLUMN_QUEUE)/* 获取 队列
ID*/
            || !row.containsKey(COLUMN_ENQPORT)/* 获取 端口
ID*/
            || !row.containsKey(COLUMN_SERVICE)){
            logger.error("Skipping entry with required fields {}", row);
/*获取服务类型*/
            continue;
        }
    }

```

4.2 CLI 指令配置模块实现

模块的具体实现将在后续章节进行详细阐述。控制平面上模块间的交互动作如下：

- (1) CLI 指令配置模块下发管理员的配置指令；
- (2) QoS 控制器读取指令，通过流表控制程序实现流表管理机制；
- (3) QoS 读取指令，通过查询、配置接口，对底层交换机状态、端口配置、队列配置进行对应的操作。

系统转发平面由位于系统的底层，由传输结点组成，因此除了对流经网络的分类业务流进行数据的传输外，还需要对数据流进行流量控制、带宽调整、等操作，而流分类、标记机制由控制器进行控制和管理。

以添加一条 Queue 队列为例：

```

try:
    cmd = "--controller=%s:%s --type ip --src %s --dst %s --add
--name %s" % (c,cprt,src,dest,name) /*在 Linux 命令端口加入 Queue 队列*/
    print './circuitpusher.py %s' % cmd
    c_proc = subprocess.Popen('./circuitpusher.py %s' % cmd, shell=True)
    print "Process %s started to create circuit" % c_proc.pid /*circuit 创
建完成*/
    #wait for the circuit to be created
    c_proc.wait()
except Exception as e:
    print "could not create circuit, Error: %s" % str(e)

try:
    subprocess.Popen("cat circuits.json",shell=True).wait() /*把策略写入
json*/
except Exception as e:

```

```
print "Error opening file, Error: %s" % str(e)
#cannot continue without file
exit()
```

4.3 DiffServ 流量控制模块实现

控制器模块是对 Floodlight 控制功能的扩展，即通过编程实现一些预定的 QoS 配置功能。该模块位于 OpenFlow 控制器上，主要提供基于分类业务的 QoS 策略，主要包括基于 DSCP 和 IP 头多元组匹配分类策略，数据流入队策略，主要完成三个功能：

- (1) 从命令配置组件读取 QoS 流规则，对流规则进行解析，提取流分类、QoS 控制器标记和入队的流匹配字段规则；
- (2) 将流匹配规则以底层 OpenFlow 交换机可以理解的流表形式按 OpenFlow 协议进行封装；
- (3) 通过安全通道将上述包含新流表信息的控制消息下发至相关底层交换机结点。

具体实现如下所示：

(1) 流分类

流分类通过对到达数据流进行比传统五元组更高精度的 IP 包头元组匹配，即流表匹配域中的匹配字段。系统可以通过上层控制器流表推送规则来调整流分类的匹配规则。

(2) 测量和标记

主要通过配置流规则对分类后的具有某种相同特征的流设置相应的动作指令，即通过 `set_nw_tos` 动作来修改 IP 的 TOS 值，此动作需在 OpenvSwitch 入端口中执行，确保进入的数据流打上标签。核心网络结点则直接根据 IP 根据包头的 DSCP 值进行相应的基于 PHB 的分类和入队操作。

(3) 入队

数据流在进行了分类和标记处理后，入队操作则根据标记的 DSCP 值将数据流推送到交换机指定端口上的指定队列上排队等转发处理动作。入队操作通过“`enqueue=queue:enqueueport`”的格式提供管理员进行配置，`queue` 为队列编号，`enqueueport` 为 `queue` 所在的端口号。QoS 控制器模块是在 Floodlight 控制器上进行的 QoS 的扩展，由 java 语言进行开发，其中，分类的 QoS 服务的包括 QoS 服务和 QoS 策略的添加、删除、修改等。

QoS 策略则为每类 QoS 服务提供分类、标记和入队操作的匹配规则。主要包括策略编号 `policyid`、IP 包头域、交换机通用唯一标识符（Universally Unique Identifier, UUID）标示符 `dpid`（datapath id）和本条策略（`policy`）匹配的服务类型编号 `sRef` 及本条策略的优先级（`priority`），`QoSPolicy` 类的具体定义如下：

```
public class QoSPolicy {
    public long policyid;
    public String type;
    /*IP 包头域*/
    public short ethtype;
```

```

public byte protocol;
public short ingressport;
public int ipdst;
public int ipsrc;
public byte tos;
public short vlanid;
public String ethsrc;
public String ethdst;
public short tcpudpsrcport31
public short tcpudpdstport;
public String dpid; /*交换机 UUID*/
public short set_nw_tos; /*重新标记 DSCP 值*/
/*入队操作, Enqueue 1:2, 其中 1 表示 queue 队列号, 2 表示 enqueue 入
队
端口号*/
public short queue;
public short enqueueport;
/*默认情况下服务类型为 Best Effort*/
public String sRef; /*policy 策略对应服务编号 sRef*/
public short priority = 0; /*policy 策略优先级*/
}

```

QoS 策略添加函数 addPolicy 通过指令接口获得 QoSPolicy 的匹配规则，同时通过调用 flowPusher 将流规则下发到特定底层交换机，以 String 类型的 swid 作为标记，具体代码实现如下：

```

public void addPolicy(QoSPolicy policy, String swid) {
/*从 policy 结构体中获取流表修改表项*/
OFFlowMod flow = policyToFlowMod(policy);
logger.info("Adding policy-flow {} to switch {}",flow.toString(),swid);
/*将 dpid 哈希值作为流名称的唯一标识码*/
flowPusher.addFlow(policy.name+Integer.toString(swid.hashCode()),
flow,swid);
}

```

指令配置模块通过调用 addPolicy 函数分别添加流控规则实现对数据流的分类、标记和入队操作。

```

elif obj_type == "policy":
    print "Trying to add policy %s" % json
    url = "http://%s:%s/wm/qos/policy/json" % (controller,port)
    #preserve immutable
    _json = json

```

```

try:
    req = helper.request("POST",url,_json)
    print "[CONTROLLER]: %s" % req
    r_j = simplejson.loads(req)
    if r_j['status'] != "Please enable Quality of Service":
        write_add("policy",_json)
    else:
        print "[QoSPusher] please enable QoS on controller"
except Exception as e:
    print e
    print "Could Not Complete Request"
    exit(1)
helper.close_connection()
else:
    print "Error parsing command %s" % type
    exit(1)

```

现基于 Linux 操作系统的终端来完成。CLI 指令配置模块程序通过调用 QoS 控制器 API 接口和 QoS 代理 API 接口实现 QoS 的配置功能。

下面对指令配置模块为管理员提供的主要输入指令及其功能进行详细的分析：

1. 帮助指令

语法：--help

功能：显示帮助信息。提示管理员当前输入指令的功能，和可进一步扩展的指令。

2. 状态指令

语法：status [enable | disable]

功能：查看、打开或关闭管理系统 QoS 功能。当设置为 enable 时，开启系统管理功能，系统读取流规则库文件和队列规则库文件，并下发配置到对应模块。

当设置为 disable 时，关闭 QoS 功能，删除对应模块的配置列表。

3. 列举指令

语法：list [swId] [msgType]

功能：swId 为交换机编号，及控制域内可控结点的编号。msgType 可以分为两类：控制器端的 QoS 服务、QoS 策略的流表控制规则和交换机端 QoS 代理的队列配置规则。

4. 配置指令

语法：add | dele [moduleType] [cfgContent]

功能：moduleType 可以为 conType(控制器类型)或 swType(交换机类型)，与两种类型对应的 cfgContent 配置信息同样分为两类，控制器类型配置信息为 QoS 服务、QoS 策略配置信息，交换机类型为 QoS 代理端口队列配置信息。

5. 退出指令

语法：exit

功能：退出控制程序和当前指令配置界面。

第五章 OpenFlow QoS 功能测试

5.1 系统测试环境介绍

5.1.1 测试平台

物理机安装 VMware Workstation 10。下载 SDN Hub (sdnhub.org) 构建的 all-in-one tutorial VM 并导入到 VMware。这是一个预装了很多 SDN 相关的软件和工具的 64 位的 Ubuntu 12.10 虚拟机映像。内置软件和工具如下：

- SDN 控制器：Opendaylight, Ryu, Floodlight, Pox 和 Trema
- 示例代码：hub, 2 层学习型交换机和其它应用
- Open vSwitch 1.11: 支持 Openflow 1.0, 实验性的支持 Openflow 1.2 和 1.3
- Mininet: 创建和运行示例拓扑
- Eclipse 和 Maven
- Wireshark: 协议数据包分析

5.1.2 实验拓扑

通过 Mininet 的 custom 下的 Python 文件建立自定义拓扑

```
Terminal - root@sdnhubvm: /home/ubuntu/mininet/custom
File Edit View Terminal Tabs Help
class MyTopo( Topo ):
    "Simple topology example."

    def __init__( self ):
        "Create custom topo."

        # Initialize topology
        Topo.__init__( self )

        # Add hosts and switches
        leftHost = self.addHost( 'h1' )
        rightHost = self.addHost( 'h2' )
        rightHost1 = self.addHost( 'h3' )
        leftSwitch = self.addSwitch( 's1' )
        rightSwitch = self.addSwitch( 's2' )

        # Add links
        self.addLink( leftHost, leftSwitch )
        self.addLink( leftSwitch, rightSwitch )
        self.addLink( rightSwitch, rightHost )
        self.addLink( rightSwitch, rightHost1 )

topos = { 'mytopo': ( lambda: MyTopo() ) }
```

图 5.1 Mininet 自定义拓扑

MAC 地址 00:00:00:00:00:00:00:01 、 00:00:00:00:00:00:00:02 分别为 OpenvSwitch1 和 OpenvSwitch2，连接 OpenvSwitch1 的为服务器，提供视频流、Web 等服务，连接 OpenvSwitch2 的为 Host1,Host2。

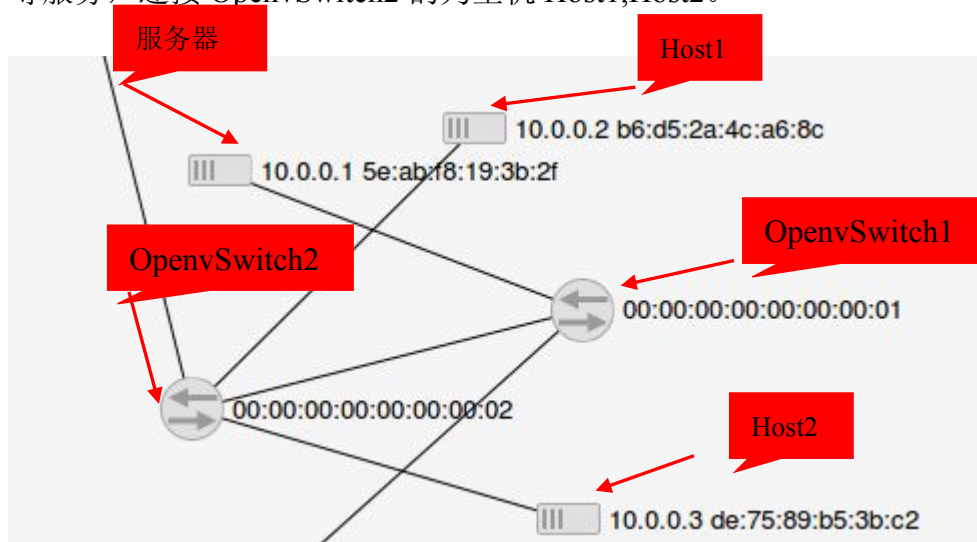


图 5.2 Floodlight 显示拓扑

5.2 实验测试方法

5.2.1 网络流量测试工具

需要一些软件辅助功能验证的执行,如用于分析数据流的网络带宽性能测试工具 iperf。

TCP 测试

客户端执行: iperf -s 是 windows 平台下命令。

服务器执行: iperf -c 10.0.0.2

5.2.2 流量控制功能验证方法

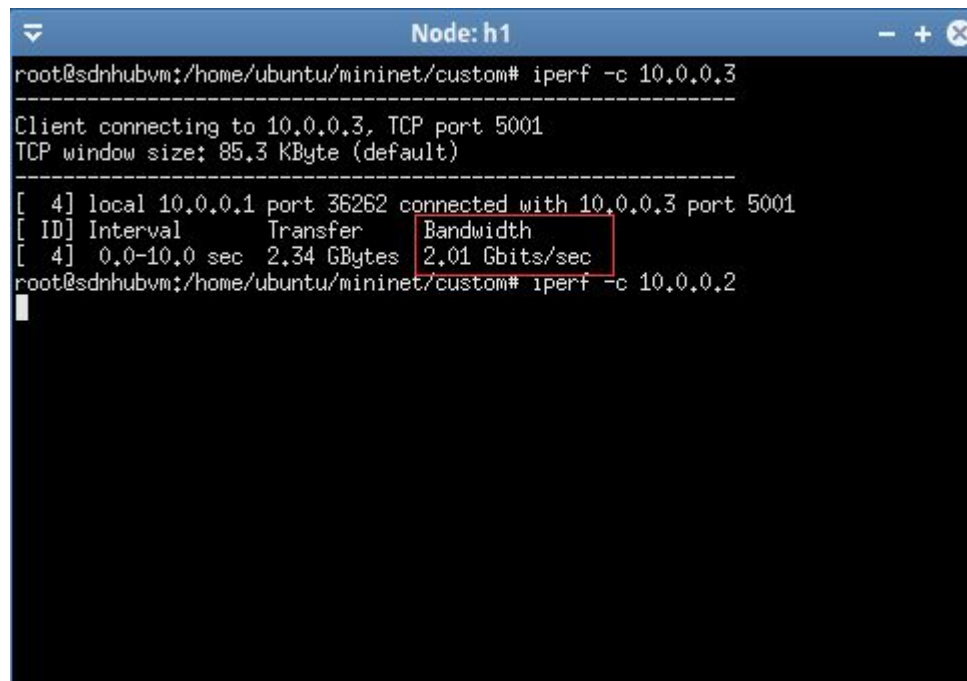
这部分实验对 OpenFlow 软交换机上 DiffServ 模块实例提供的整体 DiffServ 功能进行正确性验证。当被标记的数据流通过 OpenvSwitch 时,在其上应用的 QoS 代理对交换机进行的队列配置应具有对这些数据流的分组进行汇聚分类和转发的能力。如果 Host1 到 Host2 方向上的数据流的带宽与事先配置的 HTB 队列调度算法设置的带宽一致,则可验证 OpenFlow QoS 管理系统上的 DiffServ 模型流量控制功能的正确性。

5.3 流量控制功能验证

5.3.1 系统端口速率 TCP 限速测试

为了验证管理系统指令配置模块的配置的结果,从 Host 进行打包测试,验证配置端口速率限制的正确性。首先由服务器作为服务端,Host1 作为客户端进行 TCP 打包,然后加入 QoS 策略再进行 TCP 打包测试。从打包结果可以看出 QoS 策略完成了端口队列速率限制的功能。

首先不加入策略,服务器到 Host1 的 TCP 带宽为 2Gbps 测试如下:



```
Node: h1
root@sdnhubvm:/home/ubuntu/mininet/custom# iperf -c 10.0.0.3
-----
Client connecting to 10.0.0.3, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 4] local 10.0.0.1 port 36262 connected with 10.0.0.3 port 5001
[ ID] Interval      Transfer    Bandwidth
[ 4]  0.0-10.0 sec  2.34 GBytes 2.01 Gbits/sec
root@sdnhubvm:/home/ubuntu/mininet/custom# iperf -c 10.0.0.2
```

图 5.3 显示不加入任何 Queue 的带宽

开启 QoS 的服务成功:

```
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qosmanager2.py -e
Connection Successful
Connection Successful
Enabling QoS at 127.0.0.1:8080
[CONTROLLER]: {"status": "success", "details": "QoS Enabled"}
Closed connection successfully
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos#
```

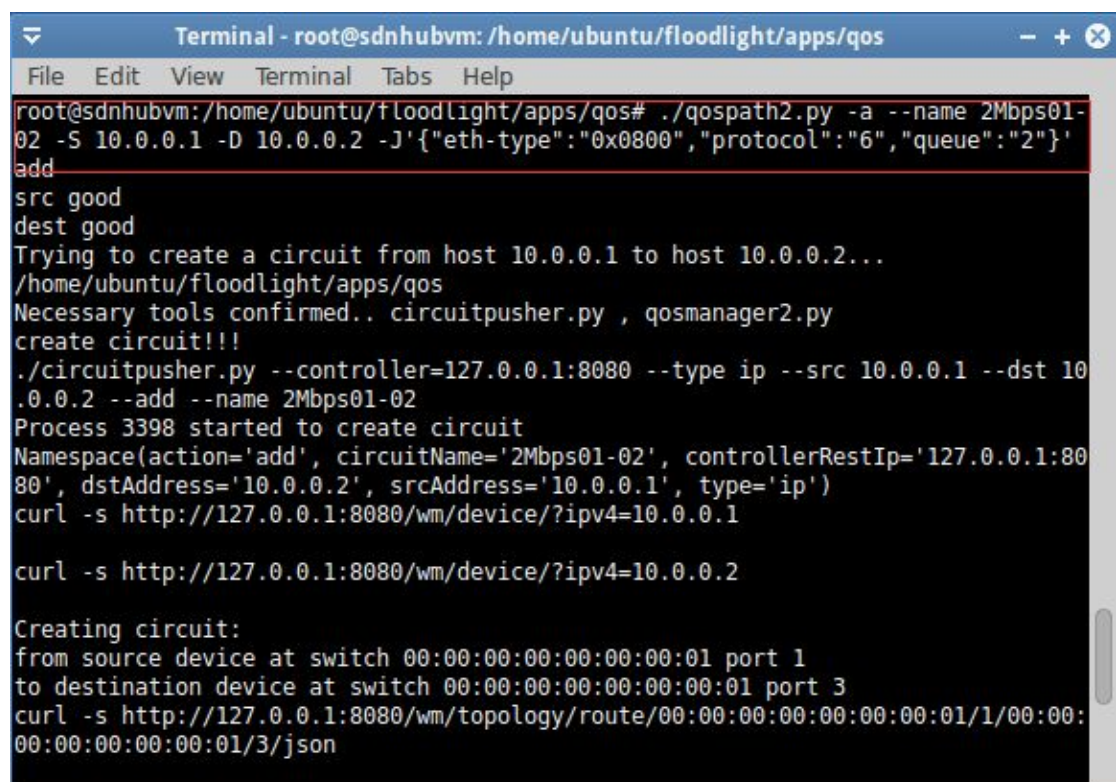
图 5.4 显示 QoS 功能开启

在 OpenvSwitch1 的接口上创建 Queue 队列机制:

```
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# sudo ovs-vsctl -- --id=@defaultq
os create qos type=linux-htb other-config:max-rate=1000000000 queues=@q0,1=@q1
,2=@q2 -- --id=@q0 create queue other-config:min-rate=1000000000 other-config:ma
x-rate=1000000000 -- --id=@q1 create queue other-config:max-rate=200000000 -- --i
d=@q2 create queue other-config:max-rate=2000000 other-config:min-rate=2000000
1b1ce4f2-65b8-4f0b-ae83-fa4edc94f13e
1f3bcda3-b3d3-4856-a7eb-f5f04241598b
8b208dbc-7697-4bee-a0f1-ddab36323bfb
963fe9d2-6aca-4aa5-b992-1435dcb8d273
```

图 5.5 OVS 的接口上创建 Queue 队列机制

创建一条实际的 QoS Policy 策略:



```
Terminal - root@sdnhubvm: /home/ubuntu/floodlight/apps/qos
File Edit View Terminal Tabs Help
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qospath2.py -a --name 2Mbps01-
02 -S 10.0.0.1 -D 10.0.0.2 -J '{"eth-type": "0x0800", "protocol": "6", "queue": "2"}'
add
src good
dest good
Trying to create a circuit from host 10.0.0.1 to host 10.0.0.2...
/home/ubuntu/floodlight/apps/qos
Necessary tools confirmed.. circuitpusher.py , qosmanager2.py
create circuit!!!
./circuitpusher.py --controller=127.0.0.1:8080 --type ip --src 10.0.0.1 --dst 10
.0.0.2 --add --name 2Mbps01-02
Process 3398 started to create circuit
Namespace(action='add', circuitName='2Mbps01-02', controllerRestIp='127.0.0.1:80
80', dstAddress='10.0.0.2', srcAddress='10.0.0.1', type='ip')
curl -s http://127.0.0.1:8080/wm/device/?ipv4=10.0.0.1

curl -s http://127.0.0.1:8080/wm/device/?ipv4=10.0.0.2

Creating circuit:
from source device at switch 00:00:00:00:00:00:01 port 1
to destination device at switch 00:00:00:00:00:00:01 port 3
curl -s http://127.0.0.1:8080/wm/topology/route/00:00:00:00:00:00:01/1/00:00:
00:00:00:00:01/3/json
```

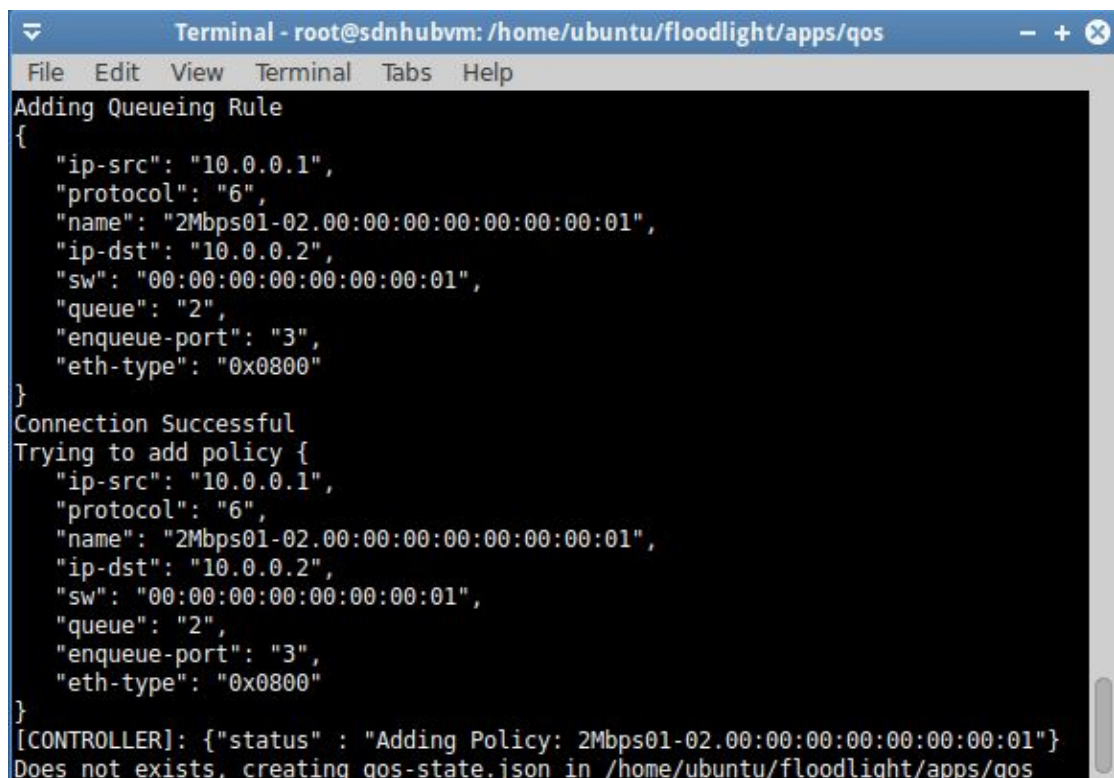
图 5.6 创建 QoS Policy

在 Floodlight 控制器中已经声明 Protocol= “6” 是 TCP 流量

```
protected static final Map<String,String> l4TypeAliasMap =
    new HashMap<String, String>();
static {
    l4TypeAliasMap.put("00", "L4_HOPOPT");
    l4TypeAliasMap.put("01", "L4_ICMP");
    l4TypeAliasMap.put("02", "L4_IGAP_IGMP_RGMP");
    l4TypeAliasMap.put("03", "L4_GGP");
    l4TypeAliasMap.put("04", "L4_IP");
    l4TypeAliasMap.put("05", "L4_ST");
    l4TypeAliasMap.put("06", "L4_TCP");
    l4TypeAliasMap.put("07", "L4_UCL");
    l4TypeAliasMap.put("08", "L4_EGP");
    l4TypeAliasMap.put("09", "L4_IGRP");
    l4TypeAliasMap.put("0a", "L4_BBN");
}
```

图 5.7 显示流量种类的 Map 表

创建 QoS Policy 策略成功，并且写入 json 文件中：



```
Terminal - root@sdnhubvm: /home/ubuntu/floodlight/apps/qos
File Edit View Terminal Tabs Help
Adding Queueing Rule
{
  "ip-src": "10.0.0.1",
  "protocol": "6",
  "name": "2Mbps01-02.00:00:00:00:00:00:00:01",
  "ip-dst": "10.0.0.2",
  "sw": "00:00:00:00:00:00:00:01",
  "queue": "2",
  "enqueue-port": "3",
  "eth-type": "0x0800"
}
Connection Successful
Trying to add policy {
  "ip-src": "10.0.0.1",
  "protocol": "6",
  "name": "2Mbps01-02.00:00:00:00:00:00:00:01",
  "ip-dst": "10.0.0.2",
  "sw": "00:00:00:00:00:00:00:01",
  "queue": "2",
  "enqueue-port": "3",
  "eth-type": "0x0800"
}
[CONTROLLER]: {"status": "Adding Policy: 2Mbps01-02.00:00:00:00:00:00:00:01"}
Does not exists, creating qos-state.json in /home/ubuntu/floodlight/apps/qos
```

图 5.8 显示 Policy

再利用 iperf 工具测试服务器到 Host1 的 TCP 速率


```
Node: h1
[ 4] local 10.0.0.1 port 56477 connected with 10.0.0.2 port 5001

[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-10.1 sec  3.62 MBytes  3.01 Mbits/sec
root@sdnhubvm:~[08:11]$
root@sdnhubvm:~[08:11]$
root@sdnhubvm:~[08:11]$ iperf -c 10.0.0.2
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 4] local 10.0.0.1 port 56478 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-10.1 sec  3.12 MBytes  2.59 Mbits/sec
root@sdnhubvm:~[08:11]$ iperf -c 10.0.0.2
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[ 4] local 10.0.0.1 port 56486 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer      Bandwidth
[ 4] 0.0-17.0 sec  256 KBytes   123 Kbits/sec
root@sdnhubvm:~[08:14]$
```

图 5.9 显示 TCP 带宽

不加入队列机制时由服务器向 Host1 发送的数据流速率在 2Gbps 左右,在加入了两条限制队列之后(一条为限制限制在 2Mbps, 另一条限制在 100kbps)

实验结果显示, 由服务器向 Host1 发送的数据流速率分别限制在 2Mbps 和 100kbps 左右, 与前面配置的预期结果一致, 证明了 QoS 系统对底层交换设备流量控制功能的正确性。

同理, 对其他流量可以做限速来保障需要额外带宽的流量。

5.3.2 系统端口 TCP 带宽保障测试

在第一个测试的基础上改变 OVS 上的 Queue 队列机制, Queue0 的机制是保障最低的带宽为 100Mbps:

```
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# sudo ovs-vsctl -- set port s1-et
h1 qos=@defaultqos -- set port s1-eth2 qos=@defaultqos -- --id=@defaultqos creat
e qos type=linux-htb other-config:max-rate=1000000000 queues=0=@q0,1=@q1,2=@q2 -
- --id=@q0 create queue other-config:min-rate=100000000 other-config:max-rate=100
0000000 -- --id=@q1 create queue other-config:max-rate=200000000 -- --id=@q2 crea
te queue other-config:max-rate=20000 other-config:mix-rate=2000
9af1a9c4-4bb6-436d-b544-e824cfa99212
7c925354-ec7e-42a5-82b5-8dbfc51a8a3f
5af709a8-ec8f-4d93-8736-0c96ba967a83
d6edc84d-0506-4163-aa70-8cdef542501f
```

图 5.10 显示 Queue0 队列

再定义一条具体的 TCP 流基于 Queue0

```
Terminal - root@sdnhubvm: /home/ubuntu/floodlight/apps/qos
File Edit View Terminal Tabs Help
Namespace(action='delete', circuitName='2Mbps01-02', controllerRestIp='127.0.0.1:8080', dstAddress='0.0.0.0', srcAddress='0.0.0.0', type='ip')
{'u'outPort': 3, 'u'Dpid': u'00:00:00:00:00:00:00:01', 'u'name': u'2Mbps01-02', 'u'inPort': 1, 'u'datetime': u'Thu May 28 08:21:29 2015'} 00:00:00:00:00:00:00:01
curl -X DELETE -d '{"name":"00:00:00:00:00:00:00:01.2Mbps01-02.f", "switch":"00:00:00:00:00:00:00:01"}' http://127.0.0.1:8080/wm/staticflowentrypusher/json
Traceback (most recent call last):
  File "./circuitpusher.py", line 190, in <module>
    print command, result
NameError: name 'result' is not defined
./qosmanager2.py --delete --type policy --json '{"policy-id":"2119571763"}' -c 127.0.0.1 -p 8080
Connection Successful
Trying to delete policy {"policy-id":"2119571763"}
comparing 2119571763 : 2119571763
[CONTROLLER]: {"status": "Type Of Service Service-ID: 2119571763 Deleted"}
Opening qos-state.json in /home/ubuntu/floodlight/apps/qos
Deleting policy from qos.state.json
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qospath2.py -a --name ,MIN-2Gbps01-02 -S 10.0.0.1 -D 10.0.0.2 -J '{"eth-type":"0x0800","protocol":"6","queue":"0"}'
```

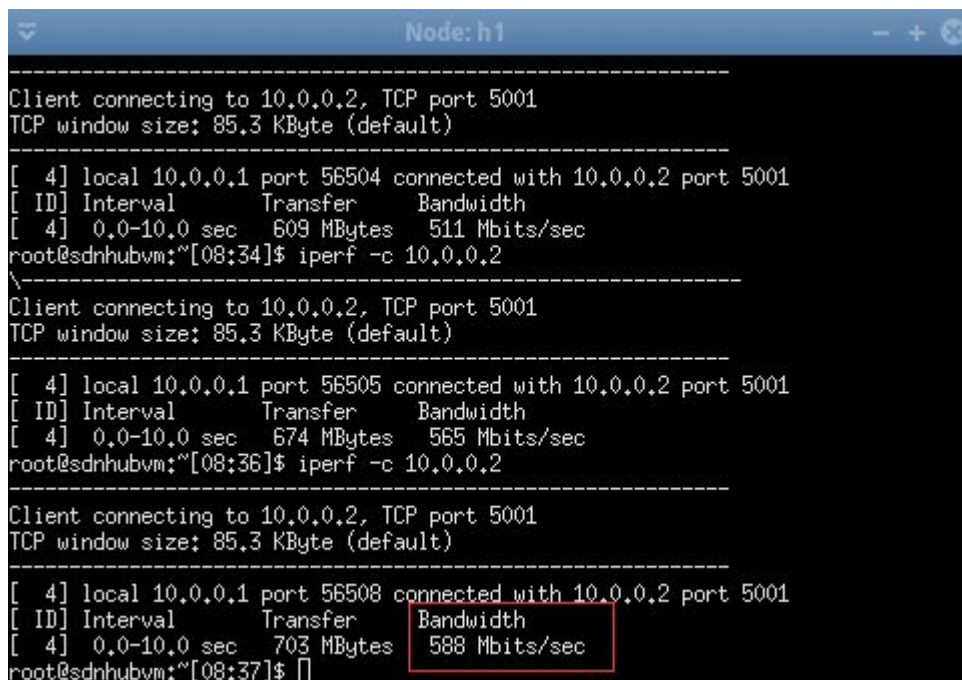
图 5.11 定义 TCP 流基于 Queue0 队列

将具体的 TCP 流基于 Queue0 的 QoS 策略写入 Json 文件

```
Terminal - root@sdnhubvm: /home/ubuntu/floodlight/apps/qos
File Edit View Terminal Tabs Help
{
  "ip-src": "10.0.0.1",
  "protocol": "6",
  "name": ",MIN-2Gbps01-02.00:00:00:00:00:00:00:01",
  "ip-dst": "10.0.0.2",
  "sw": "00:00:00:00:00:00:00:01",
  "queue": "0",
  "enqueue-port": "3",
  "eth-type": "0x0800"
}
Connection Successful
Trying to add policy {
  "ip-src": "10.0.0.1",
  "protocol": "6",
  "name": ",MIN-2Gbps01-02.00:00:00:00:00:00:00:01",
  "ip-dst": "10.0.0.2",
  "sw": "00:00:00:00:00:00:00:01",
  "queue": "0",
  "enqueue-port": "3",
  "eth-type": "0x0800"
}
[CONTROLLER]: {"status": "Adding Policy: ,MIN-2Gbps01-02.00:00:00:00:00:00:00:01"}
Does not exists, creating qos-state.json in /home/ubuntu/floodlight/apps/qos
```

图 5.12 Policy 写入 Json 文件

使用 iperf 进行测试带宽：



```
Node: h1
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[  4] local 10.0.0.1 port 56504 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[  4]  0.0-10.0 sec  609 MBytes  511 Mbits/sec
root@sdnhubvm:~[08:34]$ iperf -c 10.0.0.2
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[  4] local 10.0.0.1 port 56505 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[  4]  0.0-10.0 sec  674 MBytes  565 Mbits/sec
root@sdnhubvm:~[08:36]$ iperf -c 10.0.0.2
-----
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 85.3 KByte (default)
-----
[  4] local 10.0.0.1 port 56508 connected with 10.0.0.2 port 5001
[ ID] Interval      Transfer    Bandwidth
[  4]  0.0-10.0 sec  703 MBytes  588 Mbits/sec
root@sdnhubvm:~[08:37]$
```

图 5.13 显示 TCP 流带宽

在加入 Queue0 队列之后速度比之前 2Gbps 降低,但是 Queue0 的策略是保障最低带宽（100Mbps），所以带宽还是达到了 500Mbps。达到题目的要求。

5.3.3 系统视频流速率带宽保障测试(具体到视频流)

在 Floodlight 控制器中已经声明 Protocol= “4b” 是 Packet_Video 流量,说明可以具体到特定视频流的带宽保障。

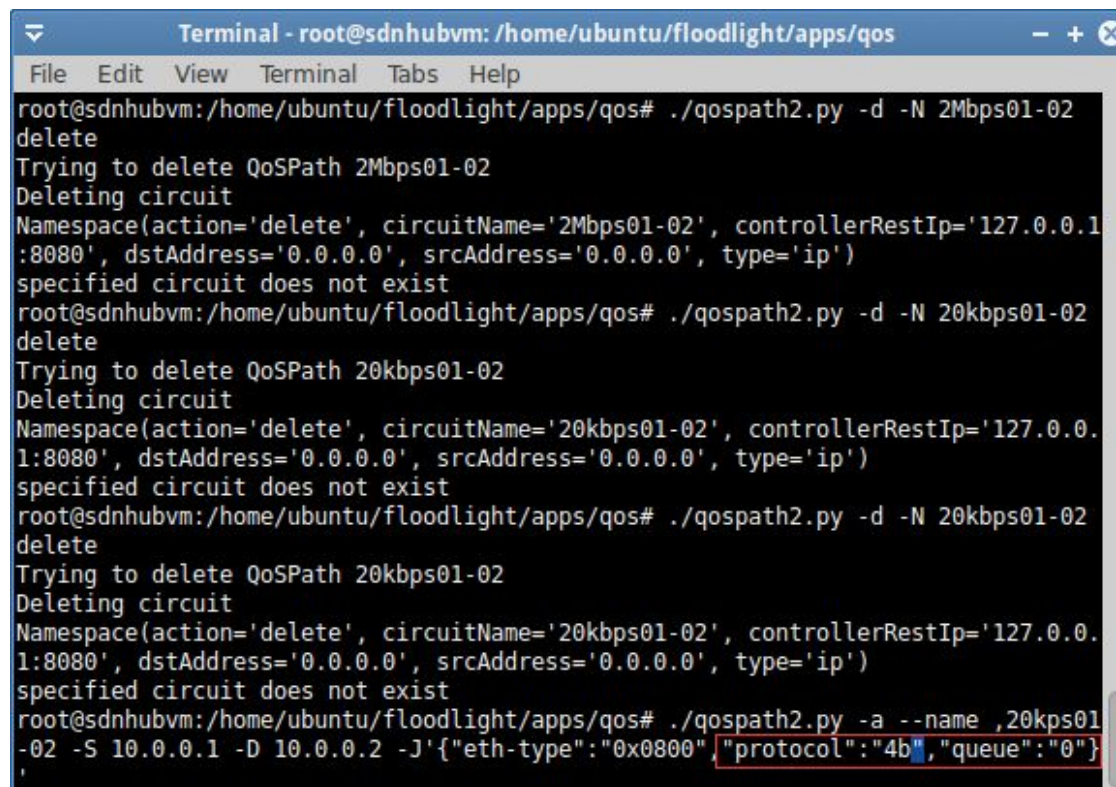
该流量类型在 Floodlight 控制器的 Counter 模块中定义。



```
l4TypeAliasMap.put("43", "L4_Internet_Pluribus");
l4TypeAliasMap.put("44", "L4_Distributed_FS");
l4TypeAliasMap.put("45", "L4_SATNET");
l4TypeAliasMap.put("46", "L4_VISA");
l4TypeAliasMap.put("47", "L4_IP_Core");
l4TypeAliasMap.put("4a", "L4_Wang_Span");
l4TypeAliasMap.put("4b", "L4 Packet_Video");
l4TypeAliasMap.put("4c", "L4_Backroom_SATNET");
l4TypeAliasMap.put("4d", "L4_SUN_ND");
l4TypeAliasMap.put("4e", "L4_WIDEBAND_Monitoring");
l4TypeAliasMap.put("4f", "L4_WIDEBAND_EXPAK");
l4TypeAliasMap.put("50", "L4_ISO_IP");
l4TypeAliasMap.put("51", "L4_VMTD");
```

图 5.14 显示视频流 Map 表

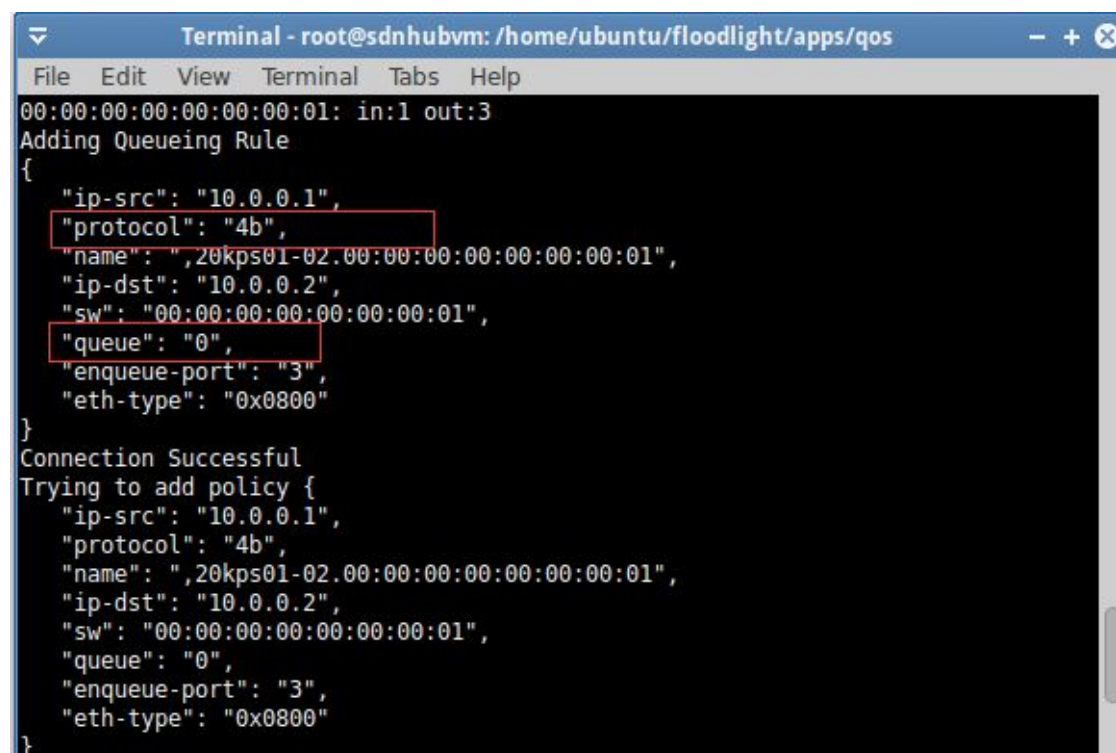
将该流量写入 QoS Policy 策略机制使用 Queue0 带宽保障队列:



```
Terminal - root@sdnhubvm: /home/ubuntu/floodlight/apps/qos
File Edit View Terminal Tabs Help
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qospath2.py -d -N 2Mbps01-02
delete
Trying to delete QoSPath 2Mbps01-02
Deleting circuit
Namespace(action='delete', circuitName='2Mbps01-02', controllerRestIp='127.0.0.1:8080', dstAddress='0.0.0.0', srcAddress='0.0.0.0', type='ip')
specified circuit does not exist
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qospath2.py -d -N 20kbps01-02
delete
Trying to delete QoSPath 20kbps01-02
Deleting circuit
Namespace(action='delete', circuitName='20kbps01-02', controllerRestIp='127.0.0.1:8080', dstAddress='0.0.0.0', srcAddress='0.0.0.0', type='ip')
specified circuit does not exist
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qospath2.py -d -N 20kbps01-02
delete
Trying to delete QoSPath 20kbps01-02
Deleting circuit
Namespace(action='delete', circuitName='20kbps01-02', controllerRestIp='127.0.0.1:8080', dstAddress='0.0.0.0', srcAddress='0.0.0.0', type='ip')
specified circuit does not exist
root@sdnhubvm:/home/ubuntu/floodlight/apps/qos# ./qospath2.py -a --name ,20kps01-02 -S 10.0.0.1 -D 10.0.0.2 -J '{"eth-type": "0x0800", "protocol": "4b", "queue": "0"}'
```

图 5.15 定义视频流

视频流 QoS Policy 写入成功:



```
Terminal - root@sdnhubvm: /home/ubuntu/floodlight/apps/qos
File Edit View Terminal Tabs Help
00:00:00:00:00:00:00:01: in:1 out:3
Adding Queueing Rule
{
  "ip-src": "10.0.0.1",
  "protocol": "4b",
  "name": ",20kps01-02.00:00:00:00:00:00:00:01",
  "ip-dst": "10.0.0.2",
  "sw": "00:00:00:00:00:00:00:01",
  "queue": "0",
  "enqueue-port": "3",
  "eth-type": "0x0800"
}
Connection Successful
Trying to add policy {
  "ip-src": "10.0.0.1",
  "protocol": "4b",
  "name": ",20kps01-02.00:00:00:00:00:00:00:01",
  "ip-dst": "10.0.0.2",
  "sw": "00:00:00:00:00:00:00:01",
  "queue": "0",
  "enqueue-port": "3",
  "eth-type": "0x0800"
}
```

5.4 实验总结

上述实验可证明本实验完成三个具体 QoS 策略：

- A. 限制基于 TCP 流量或者其他流量来保障服务级别高的带宽。**
- B. 直接保障基于 TCP 流量或者其他流量的带宽。**
- C. 还可以借助 Floodlight 控制器对视频流进行单独区分并且保障其带宽。**

下面对本文的主要研究内容和成果做以下的总结：

（1）本文通过简要的分析 OpenFlow 和 QoS 技术，结合 OpenFlow 网络技术和转发分离的思想和对 QoS 服务质量系统性管理的缺乏，提出了一种适用于 OpenFlow 网络的 QoS 管理机制。

（2）通过开源软交换机上进行二次开发，实现基于 OpenFlow 协议的传统 DiffServ 模型流分类、标记和入队的流表控制机制，和基于 HTB 算法的队列管理和队列调度算法的流量控制机制。

（3）将控制器和交换机进行集中式控制分布式处理的方式进行部署，设计并实现了控制灵活、扩展性高的 QoS 系统。

创新点：设计并实现基于可编程网络流表控制 QoS 系统框架模型，限制基于 TCP 流量或者其他流量来保障服务级别高的带宽。直接保障基于 TCP 流量或者其他流量的带宽。还可以借助 Floodlight 控制器对视频流进行单独区分并且保障其带宽。

但该存在的不足是无法在 Mininet 上模拟视频流。控制平面缺乏对网络流量的实时监控和资源利用率等信息的采集，不能根据网络环境提供动态的 QoS 策略，缺乏自适应性，因此更加智能和自适应的 QoS 管理体系是下一步的主要研究方向。

附录

流量种类 Map 表:

```
public class TypeAliases {
    protected static final Map<String,String> l3TypeAliasMap =
        new HashMap<String, String>();
    static {
        l3TypeAliasMap.put("0599", "L3_V1Ether");
        l3TypeAliasMap.put("0800", "L3_IPv4");
        l3TypeAliasMap.put("0806", "L3_ARP");
        l3TypeAliasMap.put("8035", "L3_RARP");
        l3TypeAliasMap.put("809b", "L3_AppleTalk");
        l3TypeAliasMap.put("80f3", "L3_AARP");
        l3TypeAliasMap.put("8100", "L3_802_1Q");
        l3TypeAliasMap.put("8137", "L3_Novell_IPX");
        l3TypeAliasMap.put("8138", "L3_Novell");
        l3TypeAliasMap.put("86dd", "L3_IPv6");
        l3TypeAliasMap.put("8847", "L3_MPLS_uni");
        l3TypeAliasMap.put("8848", "L3_MPLS_multi");
        l3TypeAliasMap.put("8863", "L3_PPPoE_DS");
        l3TypeAliasMap.put("8864", "L3_PPPoE_SS");
        l3TypeAliasMap.put("886f", "L3_MSFT_NLB");
        l3TypeAliasMap.put("8870", "L3_Jumbo");
        l3TypeAliasMap.put("889a", "L3_HyperSCSI");
        l3TypeAliasMap.put("88a2", "L3_ATA_Ethernet");
        l3TypeAliasMap.put("88a4", "L3_EtherCAT");
        l3TypeAliasMap.put("88a8", "L3_802_1ad");
        l3TypeAliasMap.put("88ab", "L3_Ether_Powerlink");
        l3TypeAliasMap.put("88cc", "L3_LLDP");
        l3TypeAliasMap.put("88cd", "L3_SERCOS_III");
        l3TypeAliasMap.put("88e5", "L3_802_1ae");
        l3TypeAliasMap.put("88f7", "L3_IEEE_1588");
        l3TypeAliasMap.put("8902", "L3_802_1ag_CFM");
        l3TypeAliasMap.put("8906", "L3_FCoE");
        l3TypeAliasMap.put("9000", "L3_Loop");
        l3TypeAliasMap.put("9100", "L3_Q_in_Q");
        l3TypeAliasMap.put("cafe", "L3_LLT");
    }

    protected static final Map<String,String> l4TypeAliasMap =
        new HashMap<String, String>();
    static {
        l4TypeAliasMap.put("00", "L4_HOPOPT");
```

```
l4TypeAliasMap.put("01", "L4_ICMP");
l4TypeAliasMap.put("02", "L4_IGAP_IGMP_RGMP");
l4TypeAliasMap.put("03", "L4_GGP");
l4TypeAliasMap.put("04", "L4_IP");
l4TypeAliasMap.put("05", "L4_ST");
l4TypeAliasMap.put("06", "L4_TCP");
l4TypeAliasMap.put("07", "L4_UCL");
l4TypeAliasMap.put("08", "L4_EGP");
l4TypeAliasMap.put("09", "L4_IGRP");
l4TypeAliasMap.put("0a", "L4_BBN");
l4TypeAliasMap.put("0b", "L4_NVP");
l4TypeAliasMap.put("0c", "L4_PUP");
l4TypeAliasMap.put("0d", "L4_ARGUS");
l4TypeAliasMap.put("0e", "L4_EMCON");
l4TypeAliasMap.put("0f", "L4_XNET");
l4TypeAliasMap.put("10", "L4_Chaos");
l4TypeAliasMap.put("11", "L4_UDP");
l4TypeAliasMap.put("12", "L4_TMux");
l4TypeAliasMap.put("13", "L4_DCN");
l4TypeAliasMap.put("14", "L4_HMP");
l4TypeAliasMap.put("15", "L4_Packet_Radio");
l4TypeAliasMap.put("16", "L4_XEROX_NS_IDP");
l4TypeAliasMap.put("17", "L4_Trunk_1");
l4TypeAliasMap.put("18", "L4_Trunk_2");
l4TypeAliasMap.put("19", "L4_Leaf_1");
l4TypeAliasMap.put("1a", "L4_Leaf_2");
l4TypeAliasMap.put("1b", "L4_RDP");
l4TypeAliasMap.put("1c", "L4_IRTP");
l4TypeAliasMap.put("1d", "L4_ISO_TP4");
l4TypeAliasMap.put("1e", "L4_NETBLT");
l4TypeAliasMap.put("1f", "L4_MFE");
l4TypeAliasMap.put("20", "L4_MERIT");
l4TypeAliasMap.put("21", "L4_DCCP");
l4TypeAliasMap.put("22", "L4_Third_Party_Connect");
l4TypeAliasMap.put("23", "L4_IDPR");
l4TypeAliasMap.put("24", "L4_XTP");
l4TypeAliasMap.put("25", "L4_Datagram_Delivery");
l4TypeAliasMap.put("26", "L4_IDPR");
l4TypeAliasMap.put("27", "L4_TP");
l4TypeAliasMap.put("28", "L4_ILTP");
l4TypeAliasMap.put("29", "L4_IPv6_over_IPv4");
l4TypeAliasMap.put("2a", "L4_SDRP");
l4TypeAliasMap.put("2b", "L4_IPv6_RH");
l4TypeAliasMap.put("2c", "L4_IPv6_FH");
```

```
l4TypeAliasMap.put("2d", "L4_IDRP");
l4TypeAliasMap.put("2e", "L4_RSVP");
l4TypeAliasMap.put("2f", "L4_GRE");
l4TypeAliasMap.put("30", "L4_DSR");
l4TypeAliasMap.put("31", "L4_BNA");
l4TypeAliasMap.put("32", "L4_ESP");
l4TypeAliasMap.put("33", "L4_AH");
l4TypeAliasMap.put("34", "L4_I_NLSP");
l4TypeAliasMap.put("35", "L4_SWIPE");
l4TypeAliasMap.put("36", "L4_NARP");
l4TypeAliasMap.put("37", "L4_Minimal_Encapsulation");
l4TypeAliasMap.put("38", "L4_TLSP");
l4TypeAliasMap.put("39", "L4_SKIP");
l4TypeAliasMap.put("3a", "L4_ICMPv6");
l4TypeAliasMap.put("3b", "L4_IPv6_No_Next_Header");
l4TypeAliasMap.put("3c", "L4_IPv6_Destination_Options");
l4TypeAliasMap.put("3d", "L4_Any_host_IP");
l4TypeAliasMap.put("3e", "L4_CFTP");
l4TypeAliasMap.put("3f", "L4_Any_local");
l4TypeAliasMap.put("40", "L4_SATNET");
l4TypeAliasMap.put("41", "L4_Kryptolan");
l4TypeAliasMap.put("42", "L4_MIT_RVDP");
l4TypeAliasMap.put("43", "L4_Internet_Pluribus");
l4TypeAliasMap.put("44", "L4_Distributed_FS");
l4TypeAliasMap.put("45", "L4_SATNET");
l4TypeAliasMap.put("46", "L4_VISA");
l4TypeAliasMap.put("47", "L4_IP_Core");
l4TypeAliasMap.put("4a", "L4_Wang_Span");
l4TypeAliasMap.put("4b", "L4_Packet_Video");
l4TypeAliasMap.put("4c", "L4_Backroom_SATNET");
l4TypeAliasMap.put("4d", "L4_SUN_ND");
l4TypeAliasMap.put("4e", "L4_WIDEBAND_Monitoring");
l4TypeAliasMap.put("4f", "L4_WIDEBAND_EXPAK");
l4TypeAliasMap.put("50", "L4_ISO_IP");
l4TypeAliasMap.put("51", "L4_VMTP");
l4TypeAliasMap.put("52", "L4_SECURE_VMTP");
l4TypeAliasMap.put("53", "L4_VINES");
l4TypeAliasMap.put("54", "L4_TTP");
l4TypeAliasMap.put("55", "L4_NSFNET_IGP");
l4TypeAliasMap.put("56", "L4_Dissimilar_GP");
l4TypeAliasMap.put("57", "L4_TCF");
l4TypeAliasMap.put("58", "L4_EIGRP");
l4TypeAliasMap.put("59", "L4_OSPF");
l4TypeAliasMap.put("5a", "L4_Sprite_RPC");
```

```
l4TypeAliasMap.put("5b", "L4_Locus_ARP");
l4TypeAliasMap.put("5c", "L4_MTP");
l4TypeAliasMap.put("5d", "L4_AX");
l4TypeAliasMap.put("5e", "L4_IP_within_IP");
l4TypeAliasMap.put("5f", "L4_Mobile_ICP");
l4TypeAliasMap.put("61", "L4_EtherIP");
l4TypeAliasMap.put("62", "L4_Encapsulation_Header");
l4TypeAliasMap.put("64", "L4_GMTP");
l4TypeAliasMap.put("65", "L4_IFMP");
l4TypeAliasMap.put("66", "L4_PNNI");
l4TypeAliasMap.put("67", "L4_PIM");
l4TypeAliasMap.put("68", "L4_ARIS");
l4TypeAliasMap.put("69", "L4_SCPS");
l4TypeAliasMap.put("6a", "L4_QNX");
l4TypeAliasMap.put("6b", "L4_Active_Networks");
l4TypeAliasMap.put("6c", "L4_IPPCP");
l4TypeAliasMap.put("6d", "L4_SNP");
l4TypeAliasMap.put("6e", "L4_Compaq_Peer_Protocol");
l4TypeAliasMap.put("6f", "L4_IPX_in_IP");
l4TypeAliasMap.put("70", "L4_VRRP");
l4TypeAliasMap.put("71", "L4_PGM");
l4TypeAliasMap.put("72", "L4_0_hop");
l4TypeAliasMap.put("73", "L4_L2TP");
l4TypeAliasMap.put("74", "L4_DDX");
l4TypeAliasMap.put("75", "L4_IATP");
l4TypeAliasMap.put("76", "L4_ST");
l4TypeAliasMap.put("77", "L4_SRP");
l4TypeAliasMap.put("78", "L4_UTI");
l4TypeAliasMap.put("79", "L4_SMP");
l4TypeAliasMap.put("7a", "L4_SM");
l4TypeAliasMap.put("7b", "L4_PTP");
l4TypeAliasMap.put("7c", "L4_ISIS");
l4TypeAliasMap.put("7d", "L4_FIRE");
l4TypeAliasMap.put("7e", "L4_CRTP");
l4TypeAliasMap.put("7f", "L4_CRUDP");
l4TypeAliasMap.put("80", "L4_SSCOPMCE");
l4TypeAliasMap.put("81", "L4_IPLT");
l4TypeAliasMap.put("82", "L4_SPS");
l4TypeAliasMap.put("83", "L4_PIPE");
l4TypeAliasMap.put("84", "L4_SCTP");
l4TypeAliasMap.put("85", "L4_Fibre_Channel");
l4TypeAliasMap.put("86", "L4_RSVP_E2E_IGNORE");
l4TypeAliasMap.put("87", "L4_Mobility_Header");
l4TypeAliasMap.put("88", "L4_UDP_Lite");
```

```
l4TypeAliasMap.put("89", "L4_MPLS");  
l4TypeAliasMap.put("8a", "L4_MANET");  
l4TypeAliasMap.put("8b", "L4_HIP");  
l4TypeAliasMap.put("8c", "L4_Shim6");  
l4TypeAliasMap.put("8d", "L4_WESP");  
l4TypeAliasMap.put("8e", "L4_ROHC");  
    }  
}
```