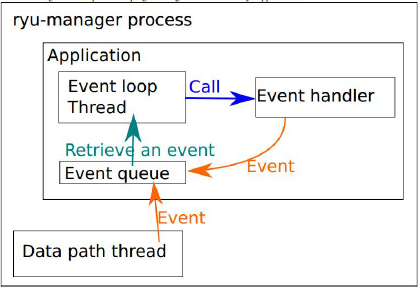
## 0 引言

SDN是一种非常先进的网络架构，可以更好地满足下一代网络的发展需要。区别于传统网络架构，它可将网络设备上的控制权分离出来，由集中的控制器管理，不依赖底层设备。然而在面对物联网（IoT）业务如智能穿戴设备、智慧城市、车联网和工业控制等，网络实时性会是重要要求。因此本文研究一种在SDN集中式控制平面基础上，应用以Floyd算法来搜索延时最优路径的思路，以达到网络实时性的全局优化目的。Floyd算法通过一个图的带权邻接矩阵搜索任意两点间最短路径矩阵，它是一种动态规划算法，稠密图效果最佳，执行效率要高于|V|次Dijkstra和SPFA算法。在研究中我们根据控制器提供的网络实时链路时延情况，计算出业务的时延优化路径，从而给SDN交换机提供数据流转发策略。本文利用SDN网络虚拟软件Mininet以及RYU控制器搭建并设置了SDN研究环境，生成自定义网络拓扑结构，通过Floyd算法进行路由搜索，进行优化SDN网络实时性能研究。

## 1 RYU、Mininet介绍

### RYU介绍



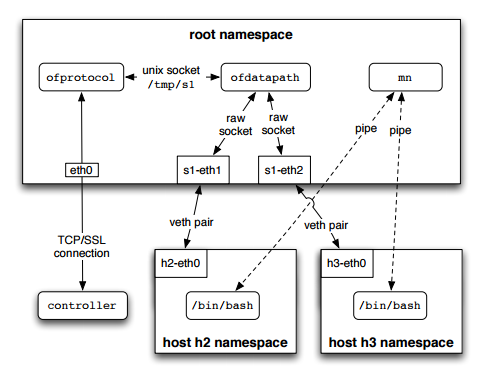
（图）RYU架构模型

Ryu的事件处理、进程与线程模块：

1. Applications：该类继承自ryu.base.app\_manager.RyuApp，用户逻辑被描述为一个APP。
2. Event：继承自ryu.controller.event.EventBase，应用程序之间的通信由transmitting and receiving events完成。
3. Event Queue：每一个application都有一个队列用于接收事件。
4. Threads：使用eventlets 库运行多线程，线程为非抢占式。
5. Event loops：创建一个application时，自动生成一个线程，该线程运行一个事件循环。
6. Additional threads：可以使用hub.spawn添加其它线程，用来处理特殊的应用
7. Eventlets：提供线程和事件队列的实现。
8. Event handlers：使用ryu.controller.handler.set\_ev\_cls 修饰一个事件处理函数。当该类型的事件触发后，事件处理函数就会被应用程序的事件循环调用。

RYU研究环境：RYU 4.9/fedora22-x64

### Mininet介绍



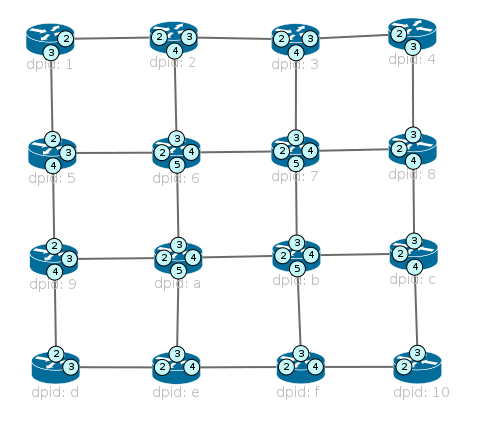
（图）Mininet框架示意

Mininet是一个轻量级软件定义网络和测试平台；它采用轻量级的虚拟化技术使一个单一的系统看起来像一个完整的网络运行。也可简单理解为SDN网络系统中的一种基于进程虚拟化平台，它支持OpenFlow等各种协议，也可以模拟一个完整的网络主机、链路和交换机在同一台计算机上，且有助于开发、测试和演示OpenFlow和SDN技术；同时也可将此进程虚拟化的平台下代码迁移到真实的环境中。

Mininet研究环境：Mininet 2.2.1/ubuntu14.04-x64

## 2 研究环境设计

### 2.1 虚拟SDN交换机网络



（图）网络拓扑

Mininet中创建16个SDN交换机构成的4X4矩阵网络，每个交换机连接一台主机。同时在交换机链路上增加1至9.9毫秒的随机时延，用以模拟实际场景中的线路。

### 2.2 测量网络链路单向时延

#### 2.2.1 时延测量原理

Controller

SwitchA

SwitchB

TechoA

TechoB

TLLDP

Tlink

（图）SDN网络时延构成示意

通过分析SDN控制器与交换机之间的时延组成，可以得知如果对链路单向时延（蓝色箭头）进行测量，则近似等于LLDP的PacketIn消息时延（绿色箭头）与相应交换机Echo消息的单向时延（红色箭头）相减，公式如下：

Tlink = TLLDP - TechoA / 2 - TechoB / 2

#### 2.2.2 TLLDP的测量

def lldp\_sent(self):

self.timestamp = time.time()

self.sent += 1

def lldp\_received(self):

self.sent = 0

delay=-1

if(self.timestamp!=None):

delay= time.time()- self.timestamp

return(delay)

……

def lldp\_packet\_in\_handler(self, ev):

……

lldpDealy=self.ports.lldp\_received(src)

TLLDP测量调整了Ryu的Switches模块代码，在lldp\_sent()中记录LLDP数据发送时的时间值，在lldp\_received()中进行当前时间值与原发送时间相减，得到了LLDP报文从控制器下发到交换机A，然后从交换机A到交换机B，再上报给控制器的时延TLLDP。实现代码如下：

#### 2.2.3 Techo的测量

Techo测量分为两个步骤。首先编辑Ryu的Controller模块代码，在\_echo\_request\_loop()中记录EchoRequest消息发送时的时间值，并存放于待发送EchoRequest消息的data字段中。实现代码如下：

def \_echo\_request\_loop(self):

while(True):

data = ('%f' % time.time()).encode('utf-8')

echo\_req = self.ofproto\_parser.OFPEchoRequest(self,data=data)

然后编辑ofp\_handler模块代码，在echo\_reply\_handler()中，利用EchoReply消息的到达时间与消息中data字段的发送时间值计算出echo消息的单向时延。实现代码如下：

def echo\_reply\_handler(self, ev):

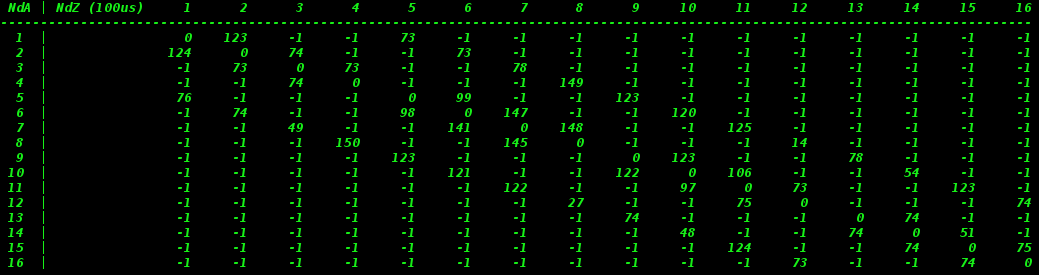
msg = ev.msg

datapath = msg.datapath

datapath.acknowledge\_echo\_reply(msg.xid)

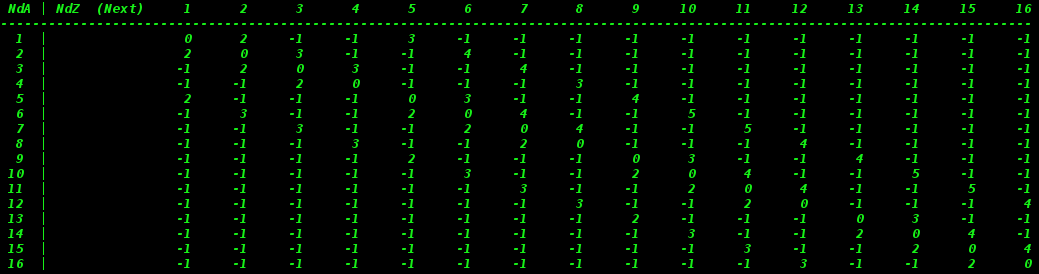
dpDelay=(int((time.time()-float(msg.data.decode('utf-8')))\*10000))>>1

### 2.3 邻接矩阵



（图）邻接时延矩阵

利用程序计算出的TLLDP和Techo值，通过公式可计算出链路时延：由于时延结果不存在负值，因此在处理非邻接链路时使用-1表示不可达；而节点到本节点的时延固定设定为0，既没有时延。其中时延数据所在行号表示源节点，数据所在列号表示目的节点，例如第一行第二个数据为123，它表示节点1到节点2的单向时延是12.3ms(123\*100us)。



（图）邻接端口矩阵

由于路径计算目的是获得在最优条件下各网络节点的下一跳转发端口，因此还需生成节点邻接端口矩阵来配合算法完成最终的转发端口矩阵。与邻接时延矩阵类似，非邻接链路端口使用-1表示不可达；而节点到本节点的邻接端口固定设定为0；端口数据所在行号表示源节点，数据所在列号表示目的节点，例如第一行第二个数据为2，它表示节点1通过端口2连接到节点2。

### 2.4 Floyd算法

Floyd算法可以解决带权有向图中任意两点间的最短路径问题，该算法的时间复杂度为O(N^3)，空间复杂度为O(N^2)，相对于dijkstra算法只能计算单点到任意点的最短路径，Floyd算法在SDN类似的网络集中控制场景更为适用。在本次研究中Floyd算法描述如下：

1. 从某个节点k开始，如果通过节点k可以使节点i到节点j的时延降低，则更新节点i到j的邻接时延矩阵，同时更新节点i到节点j的邻接端口，其值为节点i到节点k的邻接端口；
2. 节点k，i，j的遍历。

关键代码如下：

Floyd算法利用RYU实例模块中新增的一个进程，该进程主要对于邻接时延矩阵和邻接端口矩阵进行动态更新的floyd算法，周期性生成全局时延矩阵和转发矩阵。

’’’floyd算法：k-中间节点；i-起始节点；j-目的节点；nodes-网络节点总数；dm-邻接延时数组；pm-邻接端口数组

’’’

for k in range(nodes):

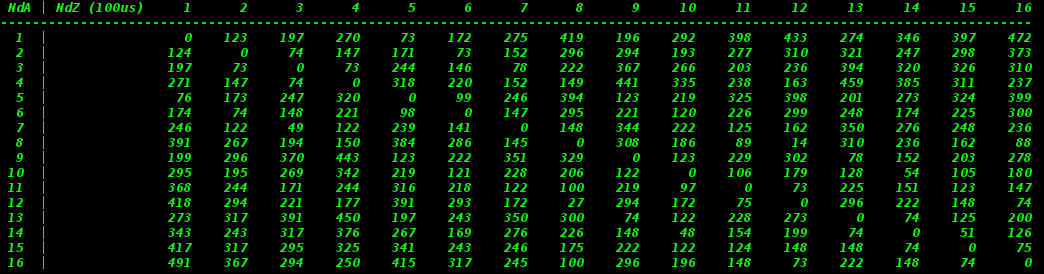
for i in range(nodes):

for j in range(nodes):

if((dm[i][k]!=-1)and(dm[k][j]!=-1)and((dm[i][k]+nm[k][j]<dm[i][j])or(dm[i][j]==-1))):

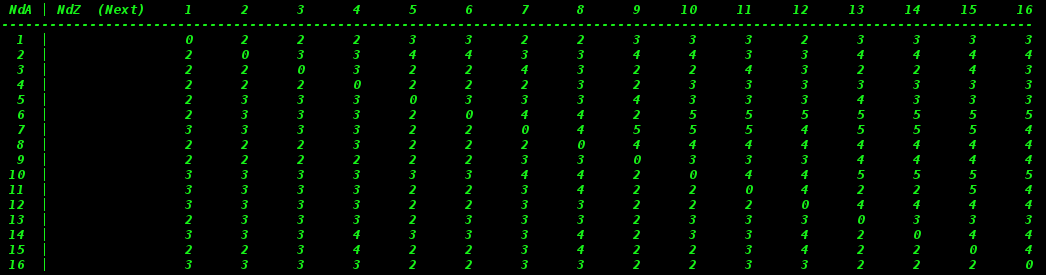
dm[i][j]=dm[i][k]+dm[k][j]

pm[i][j]=pm[i][k]



（图）全局时延矩阵

全局时延矩阵表示从节点A到节点Z的最小时延结果。



（图）全局转发矩阵

全局转发矩阵表示从节点A到节点Z的最小时延下在节点A的转发端口。

### 2.5 交换机添加流表

与传统的交换机设备不同，在SDN网络当一个报文到达openflow交换机时，会进行流表的匹配，如果没有找到相应的流表项，就会发送一个packet\_in消息到达SDN controller，控制器根据一定的路由算法决策后，向交换机下发流表(也就是发送FLOW\_MOD消息)。在研究中编辑RYU的simple\_switch\_13模块代码，新增getNextPort()以获取业务转发的下一跳端口，同时在\_packet\_in\_handler()调用以完成向SDN交换机配置流表。实现代码如下：

......

def getNextPort(self,srcDpid,dstDpid):

nextPort= pm[srcDpid-1][dstDpid-1]

return(nextPort if(nextPort==-1) else nextPort)

@set\_ev\_cls(ofp\_event.EventOFPPacketIn,MAIN\_DISPATCHER)

def \_packet\_in\_handler(self,ev):

......

if(None!=(self.macTable.get(dst,None))):

out\_port=self.getNextPort(dpid,self.macTable[dst][0])

actions=[parser.OFPActionOutput(out\_port)]

if(out\_port==None):

return()

if out\_port!=ofproto.OFPP\_FLOOD:

match=parser.OFPMatch(eth\_dst=dst)

if msg.buffer\_id!=ofproto.OFP\_NO\_BUFFER:

self.add\_flow(datapath,1,match,actions,msg.buffer\_id)

return

else:

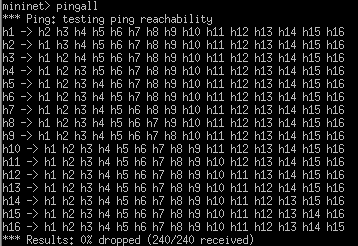
self.add\_flow(datapath,1,match,actions)

......

## 3 研究结果与分析

### 3.1 ping测试

在Mininet环境中输入pingall命令，可以测试任意两主机间ping的结果。



（图）pingall结果

根据pingall结果，研究环境中的任意两台主机间业务均可正常转发。

### 3.2 时延测试

为评估研究环境的整体时延指标，采用类似pingall的全业务遍历来统计时延数据。而从h1到h16的遍历包含总共240条业务。经过各自业务的ping测试结果，并且对比按最小跳数路径算法的结果，统计如下：

1. 全部240条业务，约有122条业务经Floyd算法优化了时延路径，优化51%的业务；
2. 优化前后全部业务的平均时延由41.6ms缩短到33.3ms，，降低20%的时延

## 4 结束语