# 基于时延和Floyd算法的SDN技术研究

熊向峰1, 许进2

（1.武汉长江通信产业集团股份有限公司,武汉 430074；2.武汉日电光通信工业有限公司，武汉 430074)

**摘要**：随着网络技术的不断演进革新，SDN已然成为一种解决现有网络问题的技术趋势。利用开放以及开源的SDN技术，对其承载的业务进行基于时延的Floyd算法方面相关研究。建立了包括Ryu和Mininet软件组成的SDN仿真实验环境，进行SDN链路延时的测量原理分析；提出以延时数据为权重的Floyd算法，并作为SDN控制器选择网络业务时延最优路径的方法；并利用仿真环境完成网络性能的评估

**关键词**：软件定义网络；RYU软件；Mininet软件；Floyd算法

# Research on SDN Technology Based on Delay and Floyd Algorithm

Xiong Xiang-feng1，Xu Jin2

(1.Wu han Yangtze Communication Industry Group CO.,LTD Wuhan 430074; 2.Wuhan NEC Fiber optic Communications Industry CO.,LTD Wuhan 430074)

**Abstract**:

**Key words**: SDN; Ryu software; Mininet software; Floyd algorithm

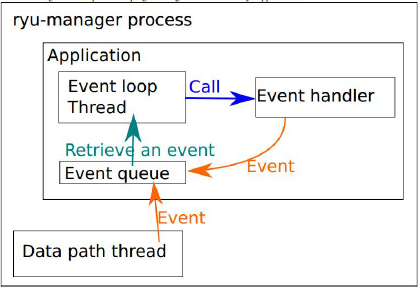
## 0 引言

SDN是一种先进的网络架构，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，网络业务时延将是一个重要的要求。因此本文研究一种在SDN的集中式控制平面基础上，应用以延时作为权重的Floyd算法来计算全局最优路径的思路，以达到网络业务时延属性的全局优化目的。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。在研究中我们将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。本此研究利用SDN开源虚拟网络软件Mininet以及SDN开源控制器RYU搭建并设置了SDN研究环境，自定义网络拓扑结构，通过基于时延的Floyd算法进行全局路径计算，进行优化SDN网络时延性能研究。

## 1 研究环境介绍

### 1.1 Ryu介绍

Ryu是由NTT公司的一款基于Python开发的开源SDN控制器，开发者或用户可以使用Python语言快速方便地实现自己的应用。Ryu 目的在于提供逻辑上的集中化管理，通过提供API使网络管理更加方便，目前RYU4.9已支持到openflow 1.4，而且已经成为著名的开源云计算平台openstack的一个可选插件。在本文中则对其架构流程进行了编辑调整，已达到研究目的。



（图）Ryu架构流程

Ryu的事件处理、进程与线程模块：

Applications：该类继承自Ryu.base.app\_manager.RyuApp，用户逻辑被描述为一个APP。

Event：继承自Ryu.controller.event.EventBase，应用程序之间的通信由transmitting and receiving events完成。

Event Queue：每一个application都有一个队列用于接收事件。

Threads：使用eventlets 库运行多线程，线程为非抢占式。

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。

Additional threads：可以使用hub.spawn添加其它线程，用来处理特殊的应用

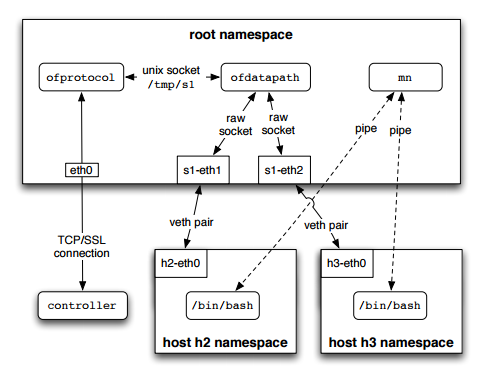
Eventlets：提供线程和事件队列的实现。

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环境：RYU 4.9/fedora22-x64

### 1.2 Mininet介绍

Mininet是stanford大学Nick McKeown研究小组基于Linux Container架构开发的一个开源SDN虚拟化平台。在Mininet中，可以方便的在个人电脑上定义且测试一个SDN环境，对基于Openflow等各种协议进行开发验证，并且轻松部署到真实的网络硬件环境中。



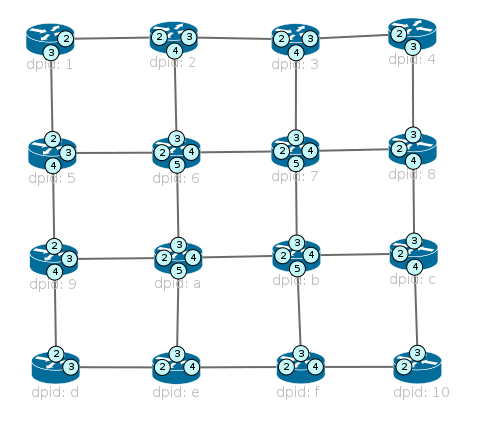
（图）Mininet框架示意

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

网络拓扑上，本次研究在Mininet中定义了16个Openflow交换机构成的4X4矩阵网络，每个交换机通过eth1连接到一台主机。同时在每段交换机链路上增加1至9.9毫秒的随机时延，以模拟物理链路上可能时延，用来虚拟一个SDN交换机网络场景。

### 2.2 测量网络链路单向时延

#### 2.2.1 时延测量原理

在Openflow协议中，链路层发现协议(LLDP) 是控制器获得SDN全网信息的关键，也是实现网络地址学习、VLAN、路由转发等网络功能的基础，SDN环境中的链路发现由控制器集中完成的，在控制器在进行链路发现时，会首先通过一个packet-out消息向所有与之连接的交换机发送LLDP数据包，然后交换机将LLDP数据包转发给相应端口，在邻居交换机将通过一个packet-in消息将LLDP数据包发送回控制器，而控制器在收到packet-in消息后，会对数据包进行分析并在其保存的链路发现表中创建这2台交换机之间的链接记录；Echo消息则是用来确认交换机与控制器之间的连接状态，在本文中将为其加入测量交换机与控制器之间时延的功能。

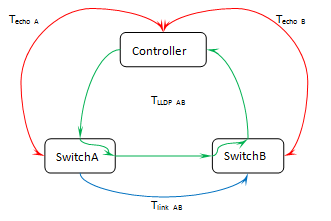
def echo\_reply\_handler(self, ev):

msg = ev.msg

datapath = msg.datapath

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

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



（图）SDN网络时延分析

通过分析SDN控制器与交换机之间的时延组成，其中TLLDP\_AB是由控制器发送经交换机链路回送到控制器的时间（绿色箭头），Techo是由控制器发送到交换机并回送到控制器的时间（红色箭头），所以可以计算得知链路单向时延（蓝色箭头）近似等于LLDP的PacketIn消息时间与相应两个交换机Echo消息时间取半相减，公式如下：

Tlink \_AB= TLLDP\_AB - Techo\_A / 2 - Techo\_B / 2

#### 2.2.2 TLLDP\_AB的测量

TLLDP\_AB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，再通过lldp\_packet\_in\_handler()进行上报。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：

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 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)

……

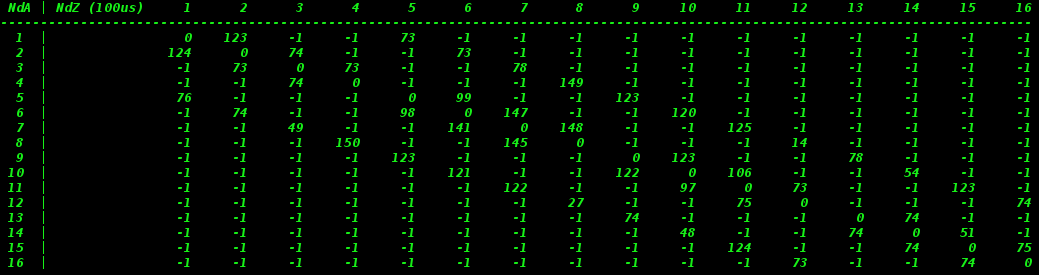
def lldp\_packet\_in\_handler(self, ev):

……

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

### 2.3 邻接矩阵

在Floyd算法中被处理的数据结构是邻接矩阵，在本文中将构建邻接时延矩阵和临界端口矩阵。



def \_echo\_request\_loop(self):

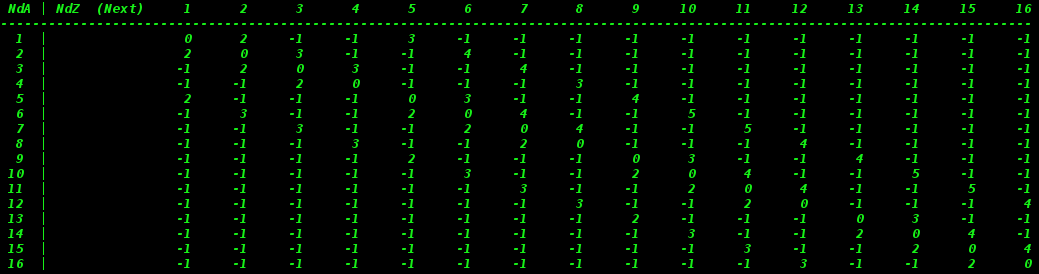
while(True):

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

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

（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算法是由Robert W. Floyd于1962年发表在“Communications of the ACM”，该算法的特点是可以解决带权有向图中任意两点间的最短路径问题，其时间复杂度为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：

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；

节点k，i，j的遍历。

关键代码如下：

由于Floyd算法较为计算密集，考虑在Python中实现，则在RYU实例模块中新增一个进程独立运行，而不会降低RYU事件消息进程的处理速度，同时可以提高多核CPU的使用效率。该进程将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端口矩阵。

’’’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):

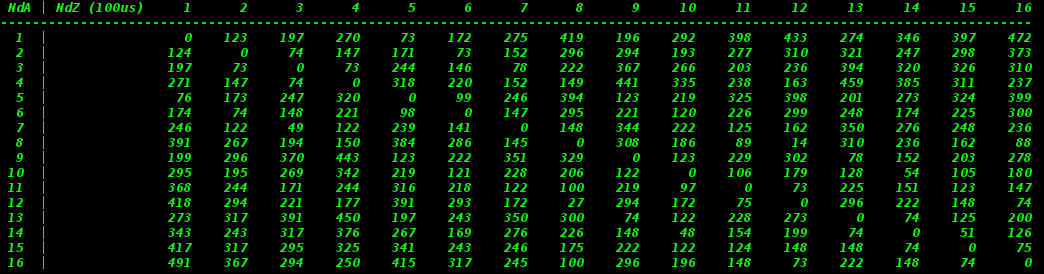
for i in range(nodes):

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))):

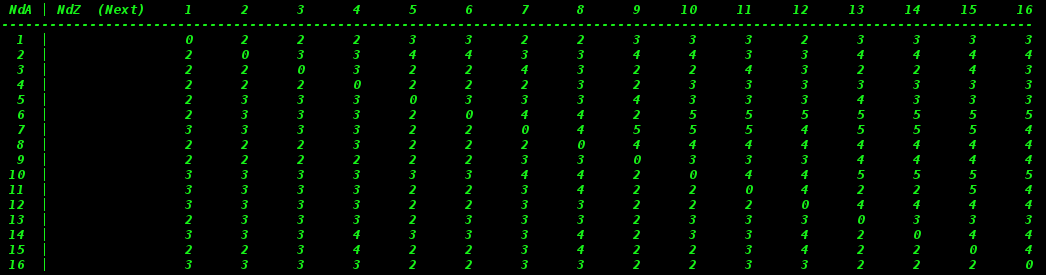
dm[i][j]=dm[i][k]+dm[k][j]

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。



......

def getNextPort(self,srcDpid,dstDpid):

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

return(nextPort if(nextPort!=-1) else None)

@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)

......

（图）全局转发端口矩阵

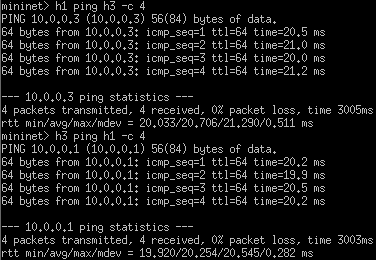
全局转发端口矩阵表示从节点A到节点Z的最小时延下在节点A的转发端口。

### 2.5 交换机流表

openflow交换机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 控制器，控制器根据预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getNextPort()以完成向SDN交换机配置流表。实现代码如下：

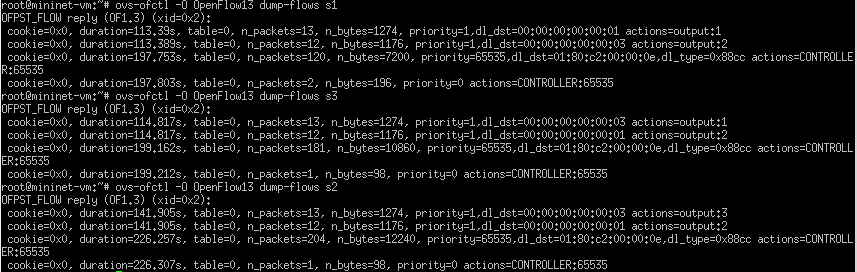
### 2.6 研究环境验证

在流表功能实现后，继而验证主机h1经s1 <-> s3到h3的互ping(如下图)，ping结果的平均时延约为20ms，这个值与Floyd算法生成的全局时延矩阵中对应的19.7ms最优结果非常接近，误差小于1ms。



（图）h1和h3互ping结果

同时检查相关联交换机的流表，与Floyd算法生成的全局转发端口矩阵的数据一致。



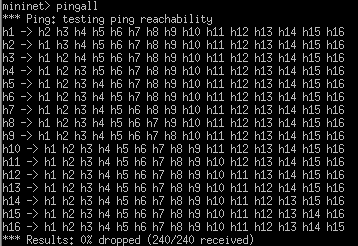
（图）s1及s2和s3的流表状态

至此研究环境搭建并初步验证完成。

## 3 研究结果与分析

### 3.1 全业务转发路径建立

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。全部业务路径根据算法结果正常配置到交换机中。

### 3.2 全业务时延测试

为评估研究环境的整体时延指标，采用ping命令进行遍历全业务来统计时延数据。而从h1到h16的遍历包含总共240条业务。通过各自业务的ping测试结果记录100次的平均时延，并与最小跳数路径算法的统计进行比较，结果如下：

全部240条业务，对比最小跳数路径有122条业务路径经过了Floyd算法优化时延，占全部业务约51%比例；

对比最小跳数路径，全业务的平均时延由41.6ms缩短到33.2ms，降低20%的时延。

## 4 结束语

在网络时延是网络的一个重要指标，它是很多网络决策的重要依据，所以网络时延测量非常重要，而SDN这样的集中式网络技术下，基于全局性的业务统筹是网络开发和维护的关键。本文通过分析Openflow的实现原理及框架，在RYU控制器中加入链路时延的测量方法和Floyd路径算法，实现了一个基本的SDN网络全局实时性业务路由方案，从而为SDN的精细化控制提供一种思路，达到提升网络性能和效率的目的。

Controller

SwitchA

SwitchB

Techo\_B

TLLDP\_AB

Tlink\_AB

Techo\_A