# SDN下组播设计与实现

段鹏飞 王磊 王少清 刘聪 单丹枫

## 总体设计思路

我们通过发特殊的控制包（dst ip为10.0.0.8的包），switch上不安装这种包的policy，所以每次遇到这种包，switch都将它转发给controller。这种包里面包含组播的host的list，controller解析这个list，用于后期的组播policy计算。

Controller的switch拓扑是通过discovery message得到的，但是controller没有办法直接得到host的拓扑结构，我们通过动态的学习host的结构，当一个新的ip出现的时候，controller会记录下来一个(ip,switch,inport)的tuple，用于保存host的拓扑信息。有了switch和host的拓扑信息之后，我们就知道了整个网络的拓扑情况。

知道了host的组播list和整个网络的拓扑之后，我们就集中计算这些host的一个最小生成树，然后所有的组播信息都在这个最小生成树上传播。

## 实现方案

我们在pyretic下实现，采用pyretic的原因主要是：

1. pyreitc提供了一个较高级的接口，可以在较高的层面上进行编程，而不用具体关心policy是具体怎么转换成flow table安装的。
2. pyretic的并行和串行模式能帮助我们解决相关的冲突。

我们把记录host拓扑和处理控制包信息的任务交给pyretic的主app，而具体根据拓扑计算最小生成树得到最终policy的过程交给了runtime本身。

因为我们初步实现的时候没有考虑到网络的link的不同，所以我们采用了很简单的计算最小生成树的算法，但是在将来，我们希望能面向load blancing和delay进行最小生成树算法的优化。

通过最小生成树得到路由信息的方法是巧妙地使用了递归的思路，如果一个switch node的孩子节点们遍历后没有生成新的有用的路由信息，并且，没有直接和这个node相连的需要进行组播的host的信息，那么这个node也不需要相关的组播路由信息。代码如下：

def travel(self,fa,v,src,mst):

print 'travel now fa is {0} node v is {1}, src is {2}'.format(fa,v,src)

po\_lst=[]

ret=false

children=[]

it=0

for port in mst.node[v]['ports'].values():

if port.possibly\_up() and port.linked\_to is not None and port.linked\_to.switch is not fa:

children.append((it,port.linked\_to.switch))

print port.linked\_to.switch

it=it+1

"""

for edge in mst.edges():

if edge[0] == v:

children.append(edge[1])

if edge[1] == v:

children.append(edge[0])

"""

for chd in children:

chd\_po\_lst=self.travel(v,chd[1],src,mst)

if(chd\_po\_lst is not []):

po\_lst.append((src,v,chd[0]))

po\_lst=po\_lst+chd\_po\_lst

for lst\_ele in self.lst\_egress\_pair:

if lst\_ele[1] == v and lst\_ele[0] in self.lst\_lst:

po\_lst.append((src,v,lst\_ele[2]))

return po\_lst

这里我们得到了一个路由信息的list，是一个(src,switch,fwd\_port)的三元组，接下来，我么将把这个三元组转化成具体的策略：

match(srcip=’’,switch=’’)>>fwd(fwd\_port)

这种转换是十分方便的，因为pyretic帮助我们隐去了policy到flow table的具体细节。

## 实现难点

本项目的最难的地方就是pyretic的结构不好，不易于实现这样复杂的应用。事实上，经过我们的调研，我们发现现在没有成熟的controller可以轻松地完成这个应用的实现。一方面是支持高级语言编程的controller不多，大部分还停留在概念的状态。对于支持高级语言编程的这些controller，扩展性太差，而且实现都很粗糙。

我们遇到的第一个问题就是，对于拓扑发现、控制包处理、最小生成树计算、生成具体policy这些任务应该在什么地方实现。我们做了很多尝试，发现把这些东西写道一个应用里几乎是不可能的（在pyretic里，因为它把很多信息分散在不同等级的api里实现），如何分配这些任务到不同的模块就是个很棘手的问题，而且最后怎么样把这些功能串起来，需要用很多的小技巧。我们在app里面实现了host拓扑的发现和控制包的处理（在app.py里面），在runtime里面实现了最小生成树的计算、根据最小生成树计算路由信息、路由信息转化为policy安装（在runtime.py）。但是最优的联调又是个很大的问题，我们给policy增加了属性，然后通过不停地反复赋值才最终实现组播的功能。而且由于python的动态语言特性，反而加大了我们的难度。

Arp的问题也影响了我们的整个app的运行，我们在发控制包之前，需要把10.0.0.8的arp信息设为静态，不然会不停地查询，迟迟不发包，这个问题调试了很长时间。

## 实验方法

首先，用户应该按照pyretic官方的要求按照pyretic并且正常运行。因为pyretic使用了POX的一些接口，实际上用户要先安装运行POX。同时用户还要安装openvswtich和mininet。这部分资料见pyretic的项目主页。

安装好了之后，请用户把我们的pyretic\_new下的代码覆盖掉pyretic的根目录。然后按照Readme.txt里面的方法运行：

运行controller：

./pyretic.py –m p0 -v please-make-it-stop pyretic.examples.app

运行mininet.sh，其中simple\_prefix是我们自己写的一个拓扑结构，如图1。

./mininet.sh –topo simple\_prefix

让host宣告自己的位置，抱住controller学习host拓扑：

pingall在mininet里面

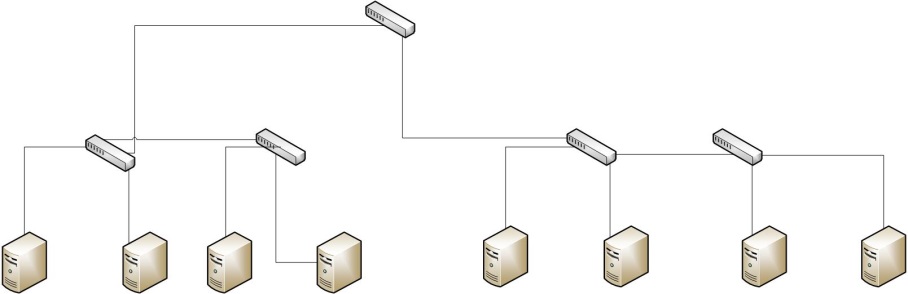


图1 我们测试采用的拓扑结构

在mininet里面调出host1的root shell

xterm h1

屏蔽控制包的arp请求：

./set\_arp.sh

发送控制包:

./app\_req.py

然后过了一会儿，就可以测试host之间的连通性了：

pingall

我们发现，组播内的host之间可以相互通信，但是之外的就不行了，验证了我们算法的可行性。