生成树机制实验报告

李昊宸

2017K8009929044

（一）生成树机制-处理config消息实现

一、实验内容

1. 基于已有代码，实现生成树运行机制，对于给定拓扑(four\_node\_ring.py)，计算输出相应状态下的最小生成树拓扑

2. 自己构造一个不少于7个节点，冗余链路不少于2条的拓扑，节点和端口的命名规则可参考four\_node\_ring.py，使用stp程序计算输出最小生成树拓扑

二、实验流程

1. 搭建实验环境

include：相关头文件

scripts：禁用TCP Offloading、IPV6功能，避免抓到无用包

main.c： Stp的代码实现，编译后在交换机结点上运行

stp.c：所有的STP机制相关，本次实验要求实现函数：

static void stp\_handle\_config\_packet(stp\_t \*stp, stp\_port\_t \*p,

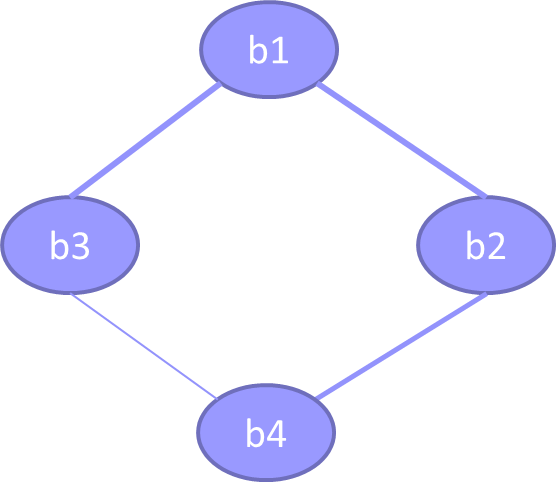
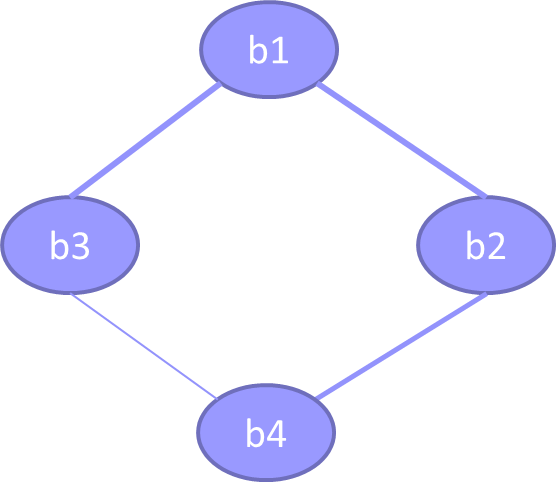
struct stp\_config \*config)

packet.c：发包函数

stp\_timer.c：定时器实现

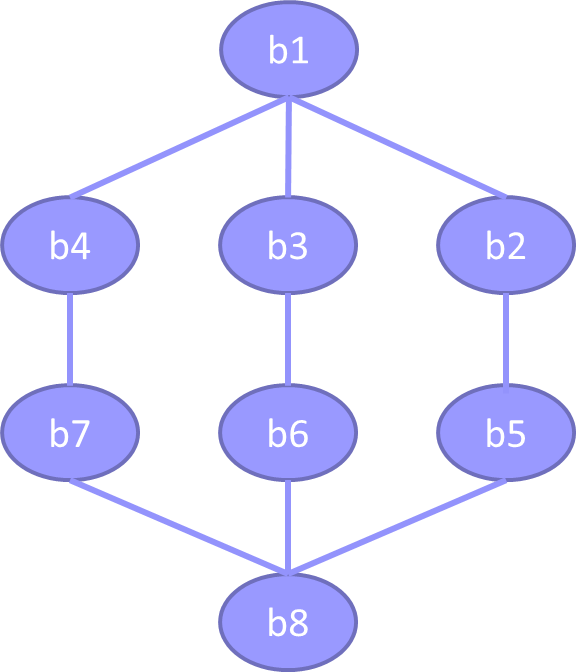
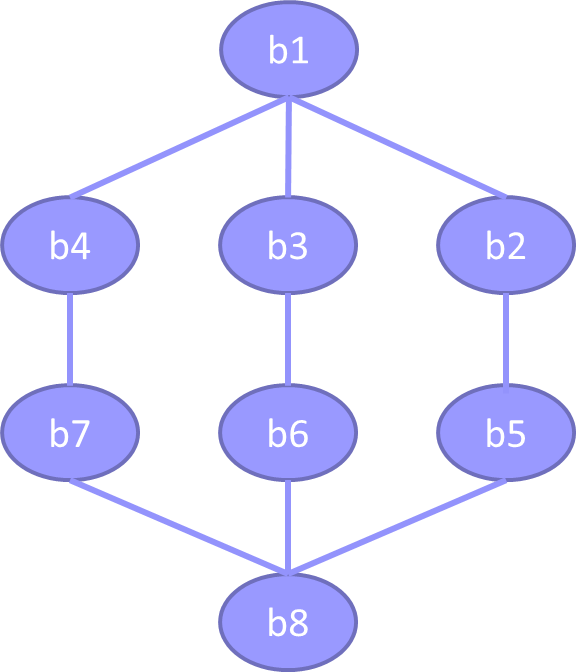
dump\_output.sh：汇总输出各节点状态信息

four\_nodes\_ring(four\_nodes\_ring \_diamond).py: 实现如下左（右）图的四节点拓扑

图一 四节点网络拓扑

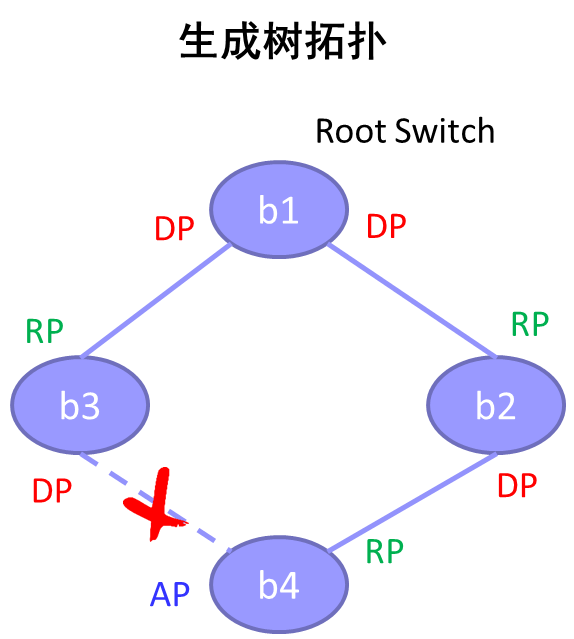
eight\_node（eight\_node\_diamond）.py：实现如下左（右）图的八节点拓扑

图二 八节点网络拓扑

生成树协议：Spanning Tree Protocol，一种用于在网络中检测环路并逻辑地阻塞冗余路径，以确保在任意两个节点之间只存在一条路径的技术。生成树机制通过禁止网络设备的相关端口，在有环路的网络中构造出一个总体开销最小的树状拓扑，使得网络在连通的前提下，避免广播风暴。

当网络中出现环路时，该协议可以采用生成树的算法从逻辑上断开其中一条连接，使其成为备份线路。当网络出现断路时，该协议会自动启动上述备份线路，确保网络正常工作。



图三 生成树拓扑

部分术语：

根节点：Root Switch，生成树网络的根。一个网络中只有一个根节点，网络中ID最小

的交换机作为根节点，也即最终网络去环后的树状图的根。

描述对象：整个网络

根端口：Root Port，RP，网络中除了根节点之外，每一个结点有一个根端口。节点在

网络中使用根端口连接到根节点，根端口是该节点到根节点路径开销最小的端

口。

描述对象：某个节点

指定端口：Designated Port，DP，每一个网段（Segment，直连网络段，不需要经由交

换机就能到达的部分，即一跳可达）有且只有一个指定端口。指定端口为该

网段所有端口中到根节点开销最小的端口。一方面，构建生成树拓扑时，STP

消息通过指定端口发送到该网段内；另一方面，该网段通过指定端口连接其

他网段。

描述对象：某个网段

其他端口：Alternate Port，AP，剩余的端口，不参与构建生成树拓扑，不转发任何消

息。

配置消息：BPDU Config，节点之间通过交换Config消息获取路径及优先级等信息。

每个端口独立生成自己的Config消息，包括：自己的节点ID，发送端口ID，

自己认为的根节点ID，以及到根节点的路径和开销。

发送方式：基于二层组播，目的MAC地址为01-80-C2-00-00-00

这里简述一下几个典型的组播地址：

01-80-C2-00-00-00(STP协议使用)

01-80-C2-00-00-01(MAC Control的PAUSE帧使用)

01-80-C2-00-00-02(Slow Protocol: 802.3ah OAM/ LACP 协议使用)

01-00-5E-xx-xx-xx(IP组播地址对应的二层组播地址)。

对于01-80-C2-00-00-00该地址，所有参与STP计算的交换机都会监听该

地址，并对目的地址为该地址的数据包进行响应。

Config消息由根节点以Hello Time的周期发出，消息老化时间为Max Age

生成树协议的数据包，从上到下封装为：IEEE 802.3 Ethernet > Logical-

Link Control > Spanning Tree Protocol

802.1D STP的消息格式：



Proto ID: STP协议标识，为0

Version: STP版本号，为0

Msg Type: 标识是配置包(0x00)还是拓扑变动包(0x80)

Flags: 标志位，第1位标识拓扑变更，第8位标志拓扑变更确认

Root Switch ID: 该节点认为的根节点ID，前16位为优先级，后48位

为MAC地址

Root Path Cost: 从该节点该端口到根节点的开销

Switch ID: 发送该消息的节点ID，定义方式同Root Switch ID

Port ID: 发送该消息的端口ID，前8位为优先级，后8位为编号

Msg Age: 该消息已存活时间，单位为1/256秒

Max Age: 消息最长允许存活时间，单位同上，默认20秒

Hello Time: 配置消息发送时间间隔，单位同上，默认2秒

Forward Delay: 不同状态间切换时延，单位同上，默认15秒

生成树协议原理：

数据结构：

typedef struct **stp** stp\_t;

struct **stp** {

    u64 switch\_id;  *// 节点ID*

    u64 designated\_root;*// 节点自己认为的根节点，指定根节点*

    int root\_path\_cost;*// 从自己到根节点的路径开销*

    stp\_port\_t \*root\_port;*// 根端口，起初该指针为空*

    long long int last\_tick;*// switch timers*

    stp\_timer\_t hello\_timer;*// hello timer*

*// ports*

    int nports; *// 端口数*

    stp\_port\_t ports[STP\_MAX\_PORTS]; *// 节点的各个端口*

    pthread\_mutex\_t lock;

    pthread\_t timer\_thread;

};

struct **stp\_port** {

    stp\_t \*stp;*// 指向该端口所在节点*

    int port\_id;*// 端口ID*

    char \*port\_name;

    iface\_info\_t \*iface;

    int path\_cost;*// 通过该端口所在网段的开销，本实验中为1*

u64 designated\_root;*// 端口自己认为的根节点，指定根节点*

    u64 designated\_switch;*// 端口所在网段到根节点的上一跳节点的ID，也就是该端口所在网段指定端口所在的节点ID*

    int designated\_port;*// 端口所在网段的指定端口*

    int designated\_cost;*// 本端口所在网段到根节点的路径开销*

};

选择根节点：

1.初始化时，所有节点都认为自己是根节点：

1.1节点将自己认定的根节点修改为自己的节点ID

1.2遍历每个端口，设置每个端口为指定端口：端口将自己认定的根节点设

置为自己所在节点,自己认定到根节点的路径开销设置为0，自己所在网段到

根节点的上一跳节点的ID设置为自己所在节点，自己所在网段的指定端口设

置为自己

2.算法开始后，当节点认为自己是根节点时，周期性的（通过hello定时器）主

动向外发送Config信息，直到该节点不再认为自己是根节点为止。如果收到的

Config信息中的根节点ID比自己认为的根节点ID小，就将自己认为的根节点更

新为消息中的根节点，并转发该Config消息。一直迭代下去，直到所有节点认为

的根节点是同一节点。

选择端口状态：

除了根节点，每个节点都有一个根端口，根端口在该节点的所有端口中到根节点

的开销最小。每个网段在自己所有的端口中选择一个到根节点开销最小的端口作为指

定端口。剩余的端口都称为其他端口。

1.某节点的某端口收到Config消息后，将其与本端口的Config进行优先级比

较：

1.1如果收到的Config优先级高（收到的Config消息只可能从该端口所在

的网段的其他端口发送而来），说明该网段应该通过对方端口连接根节点：

1.1.1首先将本端口的Config按照收到的Config消息更新（修改该端

口自己认为的根节点、自己认为的到根节点的路径代价、自己认为的所

在网段的上一跳交换机，自己认为的所在网段的指定端口）

1.1.2随后更新节点状态：遍历所有非指定端口，找到优先级最高的非

指定端口（并认定该端口为根端口，因为该端口优先级最高，意味着是

距离根节点最近的端口）：

注：该步的意义在于，接收config的端口信息更新后，可能会取代

节点原来的根端口，成为该节点新的根端口

1.1.2.1如果根端口不存在（即不存在非指定端口），说明节点本身

就是根节点，就修改节点的根端口为空，节点认为的根节点为节点

自身ID，节点认为的到根节点的路径代价为0。

1.1.2.2如果根端口存在，节点就选择通过根端口连接到根节点

（此时有可能接收Config的端口优先级提高，取代了该节点原来的

根端口），更新节点状态：节点根端口修改为刚刚查找到的根端口，

节点自己认定的根节点修改为根端口自己认定的根节点，节点自己

认为到根节点的路径开销修改为根端口所在网段到根节点的路径开

销与通过端口所在网段的路径开销之和

1.1.3节点状态更新完后，更新剩余端口的Config：

1.1.3.1对于所有指定端口，更新该端口认为的根节点为节点认为的根节点，更新该端口所在网段到根节点的路径开销为节点自己到根节点的路径开销

1.1.3.2对于非指定端口，如果该端口的Config较该端口所在网段

内其他端口优先级更高，那么修改该端口为所在网段的指定端口：

更新该端口认为的根节点为节点认为的根节点，更新该端口所在网

段到根节点的路径开销为节点自己到根节点的路径开销，更新该端

口所在网段到根节点的上一跳节点ID为当前节点，更新该端口所在

网段的指定端口ID为该端口

1.1.4之后，如果节点由根节点变为非根节点，就停止hello计时器

1.1.5最后，每个指定端口将自己的Config信息从端口自身发送出去

1.2 如果收到的Config优先级比接收端口的优先级低，说明接收端口为其所

在网段的指定端口，只将该端口本身的Config消息从端口自身发送出去

2.当STP收敛后，每个网段内所有端口的配置都相同。需要注意的是，本次实验

中，不考虑拓扑变动下的生成树重构（标准STP中，当一个节点感知到链路/端口

变化后，通过发送TCN（拓扑变动提醒）数据包告知根节点，根节点确认后再重

新构建生成树），也没有考虑如何与MAC学习共存，也没有考虑快速构建生成树

附：优先级：

1. 如果两者认为的根节点ID不同，则根节点ID小的一方优先级高
2. 若相同，如果两者到根节点的路径开销不同，则开销小的一方优先级高
3. 若相同，如果两者到根节点的上一跳节点ID不同，则上一跳节点ID小的一方优先级高
4. 若相同，如果两者到根节点的上一跳端口ID不同，则上一跳端口ID小的一方优先级高

注：以上详细信息记录在个人CSDN博客下：

<https://blog.csdn.net/Therock_of_lty/article/details/106027152>

2.启动脚本

1)四节点环形拓扑STP测试

make all

sudo python four\_nodes\_ring.py / sudo python four\_nodes\_ring\_diamonds.py

mininet> xterm b1 b2 b3 b4

b1# ./stp > b1-output.txt 2>&1

b2# ./stp > b2-output.txt 2>&1

b3# ./stp > b3-output.txt 2>&1

b4# ./stp > b4-output.txt 2>&1

//注：以上四行可以在py脚本中添加下行命令替代：

node.cmd('./stp > %s-output.txt 2>&1 &' % name)

//等待一段时间后，在另一个新终端中：

sudo pkill -SIGTERM stp

./dump\_output.sh 4

//在原终端中：

mininet> quit

2)八节点拓扑STP测试

make all

sudo python eight\_nodes.py / sudo python eight\_nodes\_diamonds.py

mininet> xterm b1 b2 b3 b4 b5 b6 b7 b8

b1# ./stp > b1-output.txt 2>&1

b2# ./stp > b2-output.txt 2>&1

b3# ./stp > b3-output.txt 2>&1

b4# ./stp > b4-output.txt 2>&1

b5# ./stp > b5-output.txt 2>&1

b6# ./stp > b6-output.txt 2>&1

b7# ./stp > b7-output.txt 2>&1

b8# ./stp > b8-output.txt 2>&1

//注：以上八行可以在py脚本中添加下行命令替代：

node.cmd('./stp > %s-output.txt 2>&1 &' % name)

//等待一段时间后，在另一个新终端中：

sudo pkill -SIGTERM stp

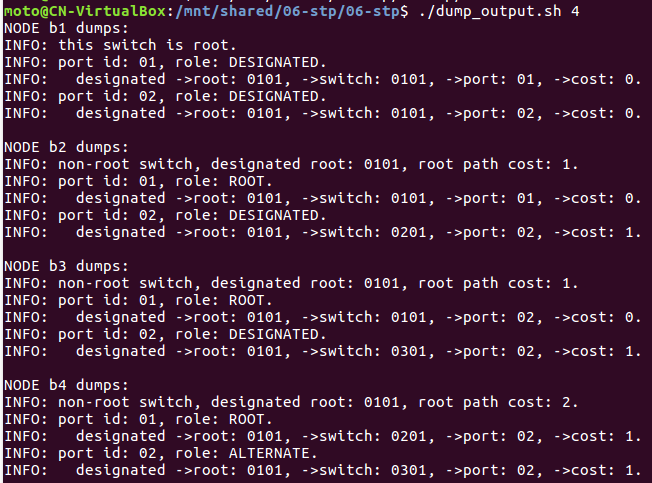
./dump\_output.sh 8

//在原终端中：

mininet> quit

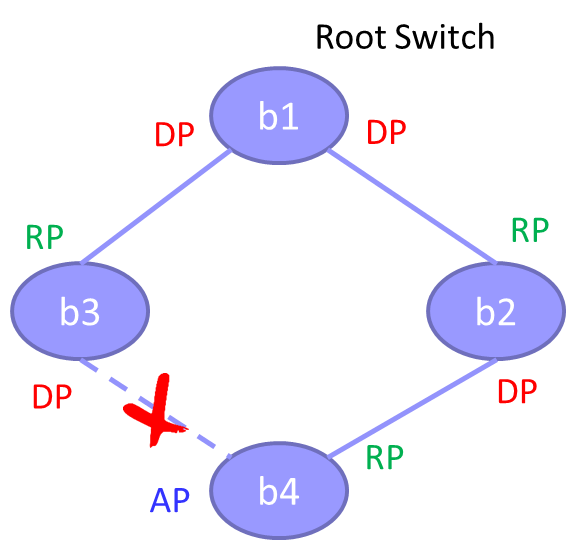
三、实验结果及分析

1. 实验结果



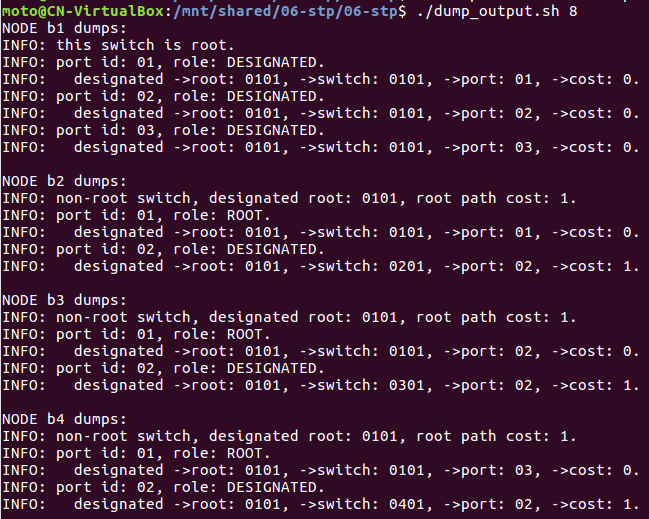
图四 四节点环型拓扑STP测试结果

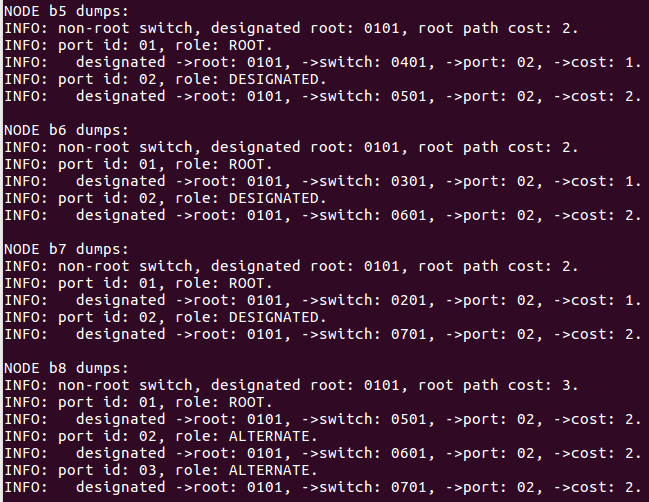
将以上信息表示成图：



图五 四节点环型拓扑STP拓扑图

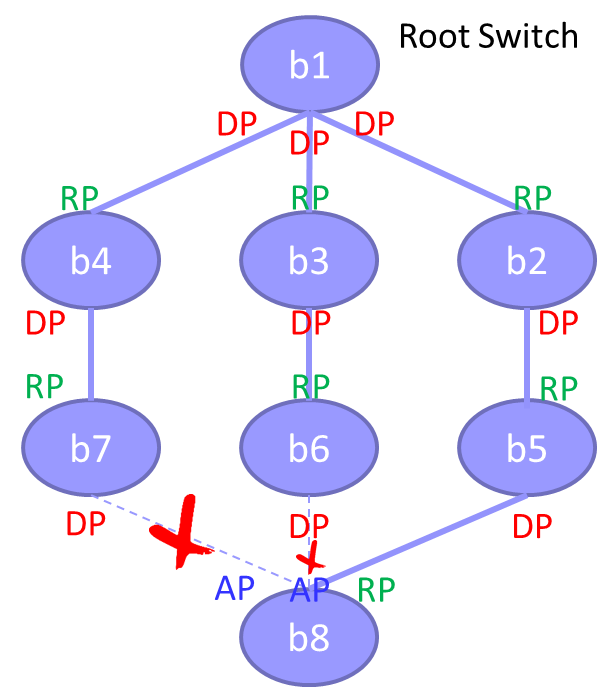
对于四节点环路，STP形成的拓扑满足我们生成优先级最高的生成树要求。





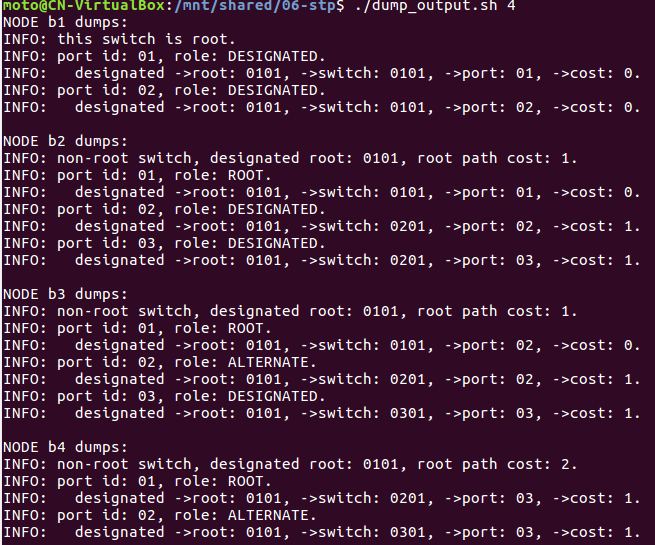
图五 八节点环型拓扑STP测试结果

将以上信息表示成图：



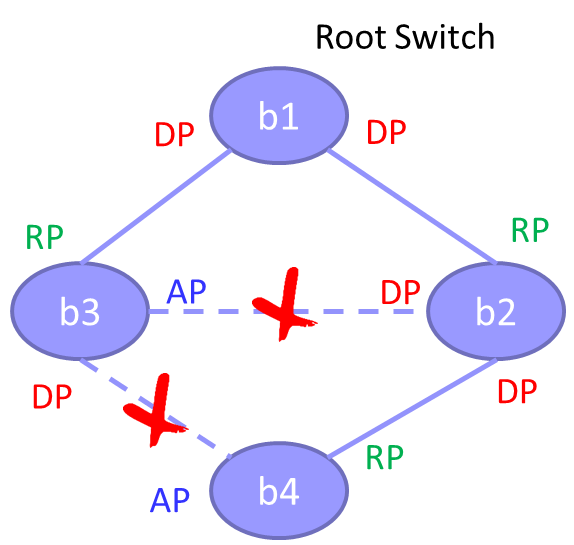
图六 八节点环型拓扑STP拓扑图

对于八节点环路，STP形成的拓扑满足我们生成优先级最高的生成树要求，尤其是对于b8而言根端口的选择，需要选择连接节点ID较小的节点。



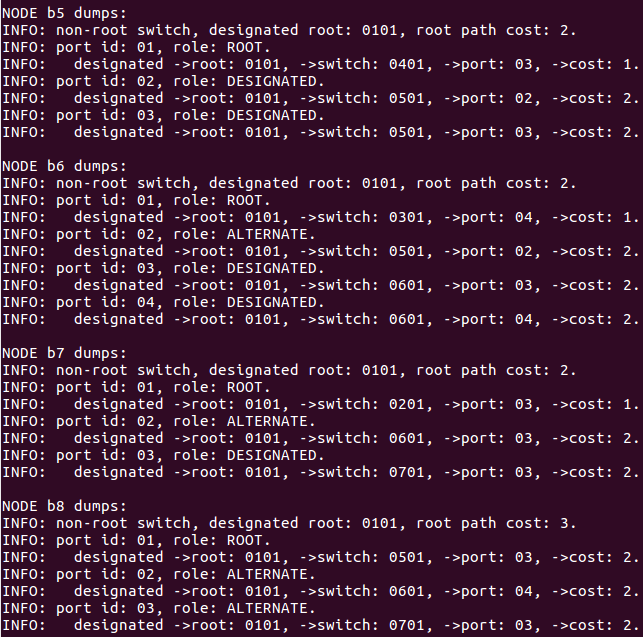
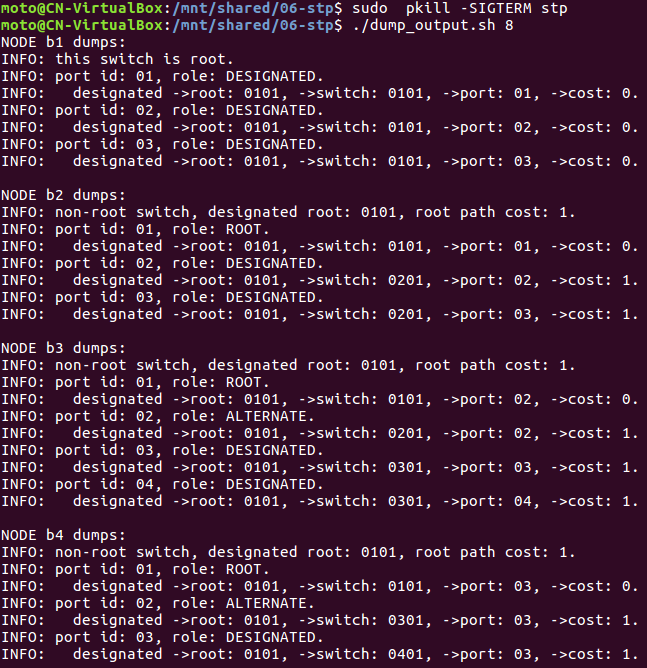
图七 四节点环型拓扑STP测试结果

将以上信息表示成图：



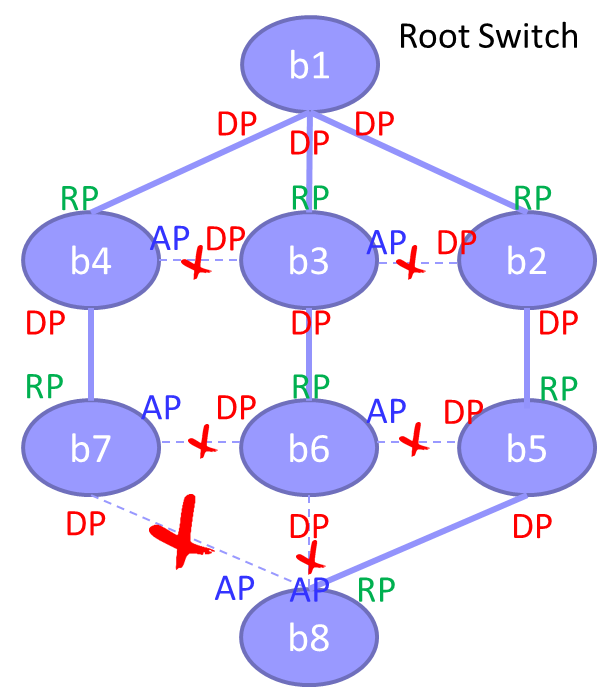
图八 四节点环型拓扑STP拓扑图

对于四节点环路，STP形成的拓扑满足我们生成优先级最高的生成树要求。特别注意的是，对于b2和b3间的网段，两个端口到根节点路径开销相同，指定端口选择节点ID较小的端口。



图九 八节点环型拓扑STP测试结果

将以上信息表示成图：



图十 八节点环型拓扑STP拓扑图

对于八节点环路，STP形成的拓扑满足我们生成优先级最高的生成树要求，一方面是对于b8而言根端口的选择，需要选择连接节点ID较小的节点；另一方面对于横向的各个网段，两个端口到根节点路径开销相同时，选择节点ID小的一个端口作为指定端口。

2. 实验分析

本次实验实现的具体代码见第二部分实验代码详解。

实现生成树协议的主函数：

int **main**(int argc, const char \*\*argv)

{

    if (**getuid**() && **geteuid**()) {

**printf**("Permission denied, should be superuser!\n");

**exit**(1);

    }

**ustack\_init**();

**ustack\_run**();

    return 0;

}

void **ustack\_init**()

{

    instance = **safe\_malloc**(sizeof(ustack\_t));

**bzero**(instance, sizeof(ustack\_t));

**init\_list\_head**(&instance->iface\_list);

**init\_all\_ifaces**();

**stp\_init**(&instance->iface\_list);

}

ustack\_init与之前几次实验比较类似，先创建实例，然后初始化实例链表和所有端口，随后调用本次实验的核心部分stp\_init函数。

void **stp\_init**(struct list\_head \*iface\_list)

{

    stp = **malloc**(sizeof(\*stp));

*// set switch ID*

    u64 mac\_addr = 0;

    iface\_info\_t \*iface = **list\_entry**(iface\_list->next, iface\_info\_t, list);

    for (int i = 0; i < sizeof(iface->mac); i++) {

        mac\_addr <<= 8;

        mac\_addr += iface->mac[i];

    }

    stp->switch\_id = mac\_addr | ((u64) STP\_BRIDGE\_PRIORITY << 48);

    stp->designated\_root = stp->switch\_id;

    stp->root\_path\_cost = 0;

    stp->root\_port = NULL;

**stp\_init\_timer**(&stp->hello\_timer, STP\_HELLO\_TIME, \

            stp\_handle\_hello\_timeout, (void \*)stp);

**stp\_start\_timer**(&stp->hello\_timer, **time\_tick\_now**());

    stp->nports = 0;

**list\_for\_each\_entry**(iface, iface\_list, list) {

        stp\_port\_t \*p = &stp->ports[stp->nports];

        p->stp = stp;

        p->port\_id = (STP\_PORT\_PRIORITY << 8) | (stp->nports + 1);

        p->port\_name = **strdup**(iface->name);

        p->iface = iface;

        p->path\_cost = 1;

**stp\_port\_init**(p);

*// store stp port in iface for efficient access*

        iface->port = p;

        stp->nports += 1;

    }

**pthread\_mutex\_init**(&stp->lock, NULL);

**pthread\_create**(&stp->timer\_thread, NULL, stp\_timer\_routine, NULL);

**signal**(SIGTERM, stp\_handle\_signal);

}

对照代码来看，首先malloc一个stp节点空间，然后list\_entry找到instance->list中的第二个节点（该节点记录了该stp节点自身的MAC信息，之所以寻找iface\_list->next，是因为iface\_list实际上是&instance->iface\_list，而该链表的第一个节点是空节点，第二个节点开始才储存信息（详情见find\_available\_ifaces函数））将记录的MAC地址和左移48位的STP\_BRIDGE\_PRIORITY（32768，默认优先级）拼接后写入到stp->switch\_id中。

随后初始化：

stp->designated\_root = stp->switch\_id;

    stp->root\_path\_cost = 0;

    stp->root\_port = NULL;

之后初始化hello计时器。

随后遍历所有的iface，为每一个iface建立一个stp\_port：

p->stp = stp;

        p->port\_id = (STP\_PORT\_PRIORITY << 8) | (stp->nports + 1);

        p->port\_name = **strdup**(iface->name);

        p->iface = iface;

        p->path\_cost = 1;

初始化每个端口：

static void **stp\_port\_init**(stp\_port\_t \*p)

{

    stp\_t \*stp = p->stp;

    p->designated\_root = stp->designated\_root;

    p->designated\_switch = stp->switch\_id;

    p->designated\_port = p->port\_id;

    p->designated\_cost = stp->root\_path\_cost;

}

将初始化后的端口保存到iface对应的port字段。最后，创建stp表项锁，启动hello线程，增加一个信号量signal(SIGTERM, stp\_handle\_signal)用于使用例外终止stp程序。

（二）实验代码详解

一、stp\_handle\_config\_packet

static void **stp\_handle\_config\_packet**(stp\_t \*stp, stp\_port\_t \*p,

        struct stp\_config \*config)

{

*// TODO: handle config packet here*

*//fprintf(stdout, "TODO: handle config packet here.\n");*

    int priority = 0;

*//compare priority between configs*

    if( p->designated\_root !=  **ntohll**(config->root\_id) )

        priority = ( p->designated\_root > **ntohll**(config->root\_id) )? 1:0;

    else if( p->designated\_cost !=  **ntohl**(config->root\_path\_cost) )

        priority = ( p->designated\_cost > **ntohl**(config->root\_path\_cost) )? 1:0;

    else if (p->designated\_switch != **ntohll**(config->switch\_id) )

        priority = ((p->designated\_switch &0xffffffffffff)> (**ntohll**(config->switch\_id) &0xffffffffffff))? 1:0;

    else if(p->designated\_port != **ntohs**(config->port\_id) )

        priority = ((p->designated\_port &0xff)> (**ntohs**(config->port\_id) &0xff))? 1:0;

首先将端口收到的config信息与端口自身的config信息进行优先级比较。

    if (priority == 1) {*//Lastest received config's priority is higher*

*// Replace config for this port*

*// This case proves that this port is designated\_port*

        p->designated\_root   = **ntohll**(config->root\_id);

        p->designated\_cost   = **ntohl** (config->root\_path\_cost);

        p->designated\_switch = **ntohll**(config->switch\_id);

        p->designated\_port   = **ntohs** (config->port\_id);

如果新收到的config优先级高，就修改端口的config

        int root\_num = 0;

        int find\_root = 1;

*// To find a root\_port.*

        for (int i = 0; i < stp->nports; i++) {

*//If one port is possiblely a root port, it is non-designated*

            if (!**stp\_port\_is\_designated**(&(stp->ports[i]))) {

                root\_num = i;

                break;

            }

*// Not found root\_port.*

            if (i == stp->nports - 1) find\_root = 0;

        }

找到第一个非指定节点，以便接下来进行根节点的查找

*//After above, we find the smallest non-designated port*

        for (int i = root\_num + 1; i < stp->nports; i++) {

            if (**stp\_port\_is\_designated**(&(stp->ports[i]))) continue;

            int priority = 1;

            if(stp->ports[i].designated\_root !=  stp->ports[root\_num].designated\_root )

                priority = ( stp->ports[i].designated\_root > stp->ports[root\_num].designated\_root )? 0:1;

            else if( stp->ports[i].designated\_cost !=  stp->ports[root\_num].designated\_cost )

                priority = ( stp->ports[i].designated\_cost > stp->ports[root\_num].designated\_cost )? 0:1;

            else if (stp->ports[i].designated\_switch != stp->ports[root\_num].designated\_switch )

*//fitst bit of switch is 1, so if compare directly, answer will be opposite*

                priority = ((stp->ports[i].designated\_switch &0xffffffffffff) > (stp->ports[root\_num].designated\_switch &0xffffffffffff))? 0:1;

            else if(stp->ports[i].designated\_port != stp->ports[root\_num].designated\_port )

                priority = ((p->designated\_port &0xff)> (stp->ports[root\_num].designated\_port &0xff))? 0:1;

            if (priority) root\_num = i;

        }

各端口比较优先级，找到config信息优先级最高的端口

*//After above, we find the root\_port(if exists)*

        if (!find\_root) {

*// This is root node.*

            stp->root\_port = NULL;

            stp->designated\_root = stp->switch\_id;

            stp->root\_path\_cost = 0;

        } else {

            stp->root\_port = &(stp->ports[root\_num]);

            stp->designated\_root = stp->root\_port->designated\_root;

            stp->root\_path\_cost = stp->root\_port->designated\_cost + stp->root\_port->path\_cost;

        }

如果没找到任何的非指定节点，说明本节点为根节点，进行节点设置

如果找到了优先级最高的节点，那么认为它为根节点，修改节点配置

*// Replace config for this node's port*

        for (int i = 0; i < stp->nports; i++) {

            if (**stp\_port\_is\_designated**(&(stp->ports[i]))) {

                stp->ports[i].designated\_root = stp->designated\_root;

                stp->ports[i].designated\_cost = stp->root\_path\_cost;

            }

*//If one net's non-designated port's designated\_cost is higher than this port*

*//'s node's root\_path\_cost, that net should change its designated port to this node's port*

            else if ((stp->root\_path\_cost < stp->ports[i].designated\_cost) || (stp->root\_path\_cost==stp->ports[i].designated\_cost)&

(stp->switch\_id < stp->ports[i].designated\_switch) ||

(stp->root\_path\_cost==stp->ports[i].designated\_cost)& (stp->switch\_id==stp->ports[i].designated\_switch)& (stp->ports[i].port\_id==stp->ports[i].designated\_port)

) { *//this judge can be added like this:*

*//*

                stp->ports[i].designated\_switch = stp->switch\_id;

                stp->ports[i].designated\_port = stp->ports[i].port\_id;

                stp->ports[i].designated\_root = stp->designated\_root;

                stp->ports[i].designated\_cost = stp->root\_path\_cost;

            }

        }

遍历所有节点，如果一个端口是指定端口，更新其认为的根节点和路径开销

如果是非指定端口，并且其config较网段内其他端口更高，那么修改该端口config使其成为指定端口，更改其认为的指定端口和节点，更新其认为的根节点和路径开销

*//If node lose its root character*

        if (!**stp\_is\_root\_switch**(stp))

**stp\_stop\_timer**(&(stp->hello\_timer));

*//Send updated config message from every desginated port*

        for (int i = 0; i < stp->nports; i++)

            if (**stp\_port\_is\_designated**(&(stp->ports[i])))

**stp\_port\_send\_config**(&(stp->ports[i]));

    } else {

**stp\_port\_send\_config**(p);

    }

}

最后，如果节点从根节点变成普通节点，就停止hello计时器，从每个指定端口发送config。

如果起初收到的config优先级没有本端口高，那么说明本端口是指定端口，从该端口发送config消息。