计算机网络研讨课实验报告（4）

裴晓坤 2019K8009918006

2022 年 4 月 18 日

一、实验题目

生成树机制实验

二、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。

三、实验流程

1. 理解实验环境

   实验所需文件及其作用列表如下：

```
scripts             # 禁止 IPv6、TCP Offloading
dump_output.sh        # 汇总输出各节点状态信息
four_node_ring.py     # 带环路网络拓扑
include             # 所有相关头文件
main.c          # 如需与 switch 实验结合，修改该文件
Makefile
device_internal.c     # 框架内部实现
stp.c           # 所有 STP 机制相关
stp-reference(.32)        # STP 参考实现
stp_timer.c           # 定时器实现
```

   config 数据结构为

```c
struct stp_config {
    struct stp_header header;
    u8 flags;           // set to 0 in this lab
    u64 root_id;        // root switch (it believes)
    u32 root_path_cost; // cost of path to root
    u64 switch_id;      // switch sending this packet
    u16 port_id;        // port sending this packet
    u16 msg_age;        // age of STP packet at tx time
    u16 max_age;        // timeout for received data: STP_MAX_AGE
    u16 hello_time;     // time between STP packet generation: STP_HELLO_TIME
    u16 fwd_delay;      // delay between states: STP_FWD_DELAY, useless in this lab
}__attribute__((packed));
```

   Port 的数据结构为

```
struct stp_port {
    stp_t *stp;                 // pointer to stp

    int port_id;                // port id
    char *port_name;
    iface_info_t *iface;

    int path_cost;              // cost of this port, always be 1 in this lab

    u64 designated_root;        // root switch (the port believes)
    u64 designated_switch;      // the switch sending this config
    int designated_port;        // the port sending this config
    int designated_cost;        // path cost to root on port
};
```

第一部分我们需要修改的文件是 stp.c 文件，第二部分需要增加一个 python 文件。

在 main.c 文件中，首先进行初始化，每个节点认为自己是根节点。

stp->designated_root = stp->switch_id

将每个端口设置为指定端口，即端口所在网段应该通过本节点连接到根节点。

p->designated_root = stp->switch_id

p->designated_cost = 0

p->designated_switch = stp->switch_id

p->designated_port = p->port_id

端口为指定端口的判断条件

p->designated_switch == stp->switch_id && p->designated_port == p->port_id

当节点认为自己是根节点时， 周期性主动发送 Config 消息。

节点通过 hello 定时器(2 秒)周期发送 Config 消息，直到该节点不再认为自己是根节点为止。

我们需要完成的部分是生成树机制运行时，处理 config 消息的算法。算法大致思路如下：

收到 Config 消息后，将其与本端口 Config 进行优先级比较，如果收到的 Config 优先级高，说明该网段应该通过对方端口连接根节点，将本端口的 Config 替换为收到的 Config 消息，本端口为非指定端口，更新节点状态，更新其余(Other)端口的 Config，如果节点由根节点变为非根节点，停止 hello 定时器，将更新后的 Config 从每个指定端口转发出去，否则，说明该网段应该通过本端口连接根节点，该端口是指定端口，发送 Config 消息。

2. 编写代码

stp.c

根据算法要求，我们可以编写处理 config 信息的函数。

```
#define get_switch_id(switch_id) (int)(switch_id & 0xFFFF)
#define get_port_id(port_id) (int)(port_id & 0xFF)
```

首先定义两个宏，用来提取节点 id 和端口 id，这是由于节点 ID 是一个 64 位整数，前 2 字节为优先级，每个节点可以独立设置优先级，默认为 32768 后 6 字节为节点第一个端口的 MAC 地址，首位为 1，导致直接比较结果可能相反。端口 ID 是一个 16 位整数，前 1 字节为优先级，可独立设置，默认为 128，后 1 字节标识该端口的序号。首位为 1，直接比较会出错，所以我们要提取出其 id 后再进行比较，该处的宏定义，参考了该文件的其他函数。

根据算法，收到 Config 消息后，将其与本端口 Config 进行优先级比较。优先级比较的过程为：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；如果两者到根节点的开销不同，则开销小的一方优先级高；如果两者到根节点的上一跳节点不同，则上一跳节点 ID 小的一方优先级高；如果两者到根节点的上一跳端口不同，则上一跳端口 ID 小的一方优先级高。

```
int priority;//1:p, 0:config
    //compare config
    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 = (get_switch_id(p->designated_switch) < get_switch_id(ntohll(config->switch_id)))?1:0;
    }else if (p->designated_port != ntohs(config->port_id))
    {
        priority = (get_port_id(p->designated_port) < get_port_id(ntohs(config->port_id)))?1:0;
    }
```

这里的 p 为本端口，config 为收到的消息，这里需要使用 ntohll、ntohl 和 ntohs 函数将网络字节序转换成主机字节序，使用 priority 来记录优先级结果。然后根据比较后的优先级进行处理。

如果收到的 config 优先级更高，说明该网段应该通过对方端口连接根节点，此时我们应该将本端口的 config 替换成收到的 config 消息。本端口为非指定端口。

```
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);
```

然后更新节点状态，遍历所有端口，找到根端口，根端口为非指定端口，端口优先级高于所有其他非指定端口。

```
//init root_num;
for (int i = 0; i < stp->nports; i++) {
    stp_port_t *p = &(stp->ports[i]);
    if (!stp_port_is_designated(p)){
        root_num = i;
        break;
    }
}
if (root_num == -1)
{
    find_foot = 0;
}

//find root port
for(int i = root_num + 1; i < stp->nports; i++){
    stp_port_t *p = &(stp->ports[i]);
    stp_port_t *root = &(stp->ports[root_num]);
    if (stp_port_is_designated(p))continue;
    int prio;//1:root_num 0:stp->ports[i]
    if(root->designated_root != p->designated_root)
    {
        prio = (root->designated_root < p->designated_root)? 1:0;
    }else if (root->designated_cost != p->designated_cost)
    {
        prio = (root->designated_cost < p->designated_cost)? 1:0;
    }else if (get_switch_id(root->designated_switch) != get_switch_id(p->designated_switch))
    {
        prio = (get_switch_id(root->designated_switch) < get_switch_id(p->designated_switch))? 1:0;
    }else if (get_port_id(root->designated_port)!=get_port_id(p->designated_port))
    {
        prio = (get_port_id(root->designated_port) < get_port_id(p->designated_port))? 1:0;
    }
    if(prio == 0){
        root_num = i;
    }
}
```

首先声明一个遍历 root_num 记录根端口的位置，用一个 for 循环来初始化，这里不能直接给 root_num 初始化，因为我们不知道哪个端口是非指定端口。所以要用循环来寻找第一个非指定端口。

之后在 stp 的 ports 数组里面遍历寻找根节点，这里仍然是比较优先级，每次两个比较，然后记录优先级最高的那个。

```
//update stp
        if(find_foot == 0)
        {
            stp->designated_root = stp->switch_id;
            stp->root_port = NULL;
            stp->root_path_cost = 0;
        }else{
            stp_port_t * root = &(stp->ports[root_num]);
            stp->root_port = root;
            stp->designated_root = root->designated_root;
            stp->root_path_cost = root->designated_cost + root->path_cost;
        }
```

如果没有找到根端口，说明节点的端口全是指定端口，所以自己本身就是根节点，于是我们只需设置参数，表明自己本身是根节点即可。

如果找到根端口，那么我们更新节点状态，让节点的根端口指向这个根端口，根节点更新为根端口认为的根节点，路径开销为根端口到根节点的开销加上根端口所在网段的开销。

根据算法，接下来我们要更新其余端口的 config。具体更新过程如下：如果一个端口为非指定端口，且其 Config 较网段内其他端口优先级更高，那么该端口成为指定端口。对于所有指定端口，更新其认为的根节点和路径开销。

```
//other config
for(int i = 0; i < stp->nports; i++){
    stp_port_t *p = &(stp->ports[i]);
    if (stp_port_is_designated(p)){
        p->designated_root = stp->designated_root;
        p->designated_cost = stp->root_path_cost;
    }else if(i!=root_num && stp->root_path_cost < stp->ports[i].designated_cost){
        p->designated_switch = stp->switch_id;
        p->designated_port = p->port_id;
        p->designated_root = stp->designated_root;
        p->designated_cost = stp->root_path_cost;
    }
}
```

这里的优先级比较与之前的不同，这里直接比较路径开销，如果从端口到根节点的开销大于通过本节点到达根节点的开销，那么我们就将该端口设置为指定端口。

```
if(!stp_is_root_switch(stp)){
        stp_stop_timer(&stp->hello_timer);
    }
    stp_send_config(stp);
```

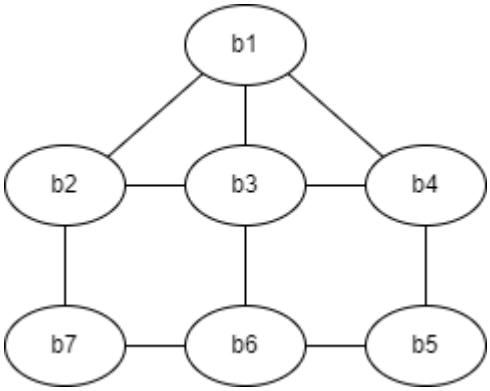
如果该节点测试由根节点变为非根节点，则停止 hello 计时器。最后将跟新后的 config 从每个端口发出去。这里直接调用封装好的函数 stp_send_config()，该函数发送 config 消息。

以上讨论的情况是收到的 config 优先级高于本端口的 config 的情况，如果低于本地 config，那么该端口为指定端口，使用 stp_port_send_config(p);发送 config 消息。

构建新的拓扑结构进行实验

```
class RingTopo(Topo):
    def build(self):
        b1 = self.addHost('b1')
        b2 = self.addHost('b2')
        b3 = self.addHost('b3')
        b4 = self.addHost('b4')
        b5 = self.addHost('b5')
        b6 = self.addHost('b6')
        b7 = self.addHost('b7')
        self.addLink(b1, b2)
        self.addLink(b1, b3)
        self.addLink(b1, b4)
        self.addLink(b2, b3)
        self.addLink(b2, b7)
        self.addLink(b3, b4)
        self.addLink(b3, b6)
        self.addLink(b4, b5)
        self.addLink(b5, b6)
        self.addLink(b6, b7)
```

构造的拓扑结构如上，使用七个节点，含有四条冗余链路结构如下所示。



3. 启动脚本

（1）四节点测试

```
make
sudo python2 four_node_ring.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
等一段时间后，在新的终端执行
sudo pkill -SIGTERM stp
./dump_output.sh 4 //输出四个节点的状态
mininet > quit
```

（2）七节点测试

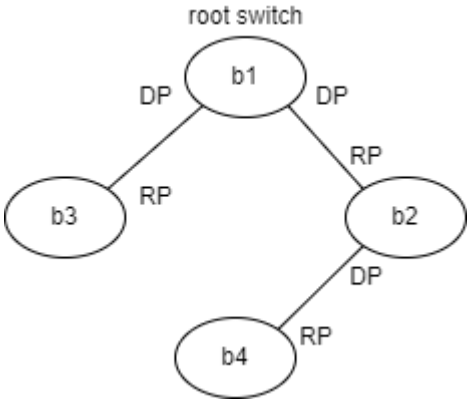
```
make
sudo python2 seven_node_ring.py
mininet> xterm b1 b2 b3 b4 b5 b6 b7
b1# ./stp > b1-output.txt 2>&1
b2# ./stp > b2-output.txt 2>&1
b3# ./stp > b3-output.txt 2>&1
b4# ./stp > b4-output.txt 2>&1
b5# ./stp > b5-output.txt 2>&1
b6# ./stp > b6-output.txt 2>&1
b7# ./stp > b7-output.txt 2>&1
等一段时间后，在新的终端执行
sudo pkill -SIGTERM stp
./dump_output.sh 7 //输出七个节点的状态
mininet > quit
```

四、实验结果及分析

1.第一部分实验测试结果：

```
stu@stu:~/cnlab/lab4/04-stp$ ./dump_output.sh 4
NODE b1 dumps:
INFO: this switch is root.
INFO: port id: 01, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.

NODE b2 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.

NODE b3 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 02, ->cost: 1.

NODE b4 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 02, ->cost: 1.
```

将以上信息绘制成图，可得



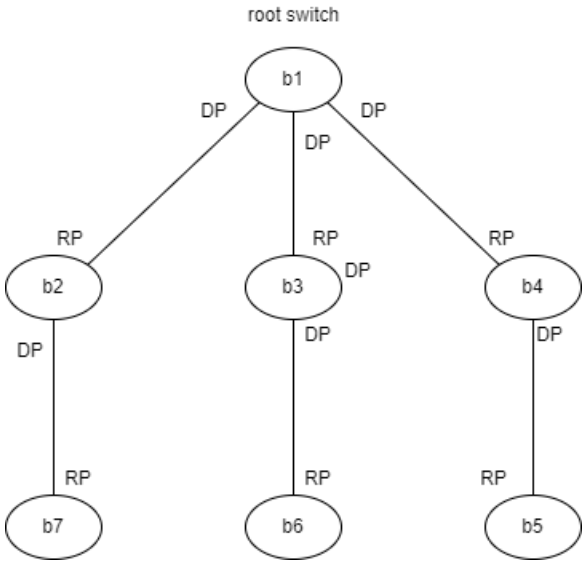
可以得出，生成树符合我们的要求。可以看出，b4 到 b1 可以经过 b3 或者 b2，但由于 b2 优先级高，所以我们选择了 b2.

2.第二部分实验测试结果：

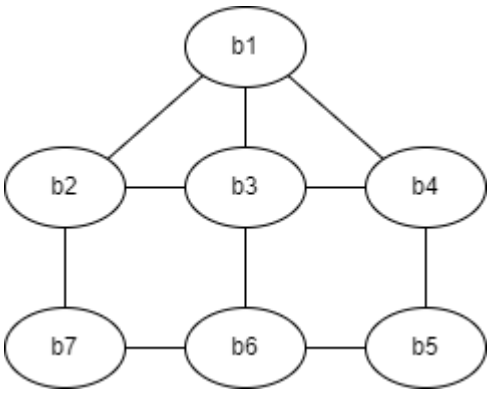
```
stu@stu:~/cnlab/lab4/04-stp$ ./dump_output.sh 7
NODE b1 dumps:
INFO: this switch is root.
INFO: port id: 01, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 03, ->cost: 0.

NODE b2 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 01, ->cost: 0.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 03, ->cost: 1.

NODE b3 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 02, ->cost: 0.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 02, ->cost: 1.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 03, ->cost: 1.
INFO: port id: 04, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 04, ->cost: 1.
```

```
NODE b4 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 1.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0101, ->port: 03, ->cost: 0.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 03, ->cost: 1.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 03, ->cost: 1.

NODE b5 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 03, ->cost: 1.
INFO: port id: 02, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0501, ->port: 02, ->cost: 2.

NODE b6 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 04, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0501, ->port: 02, ->cost: 2.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0601, ->port: 03, ->cost: 2.

NODE b7 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0201, ->port: 03, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0601, ->port: 03, ->cost: 2.
```

绘制成图，如下所示



与原图相比



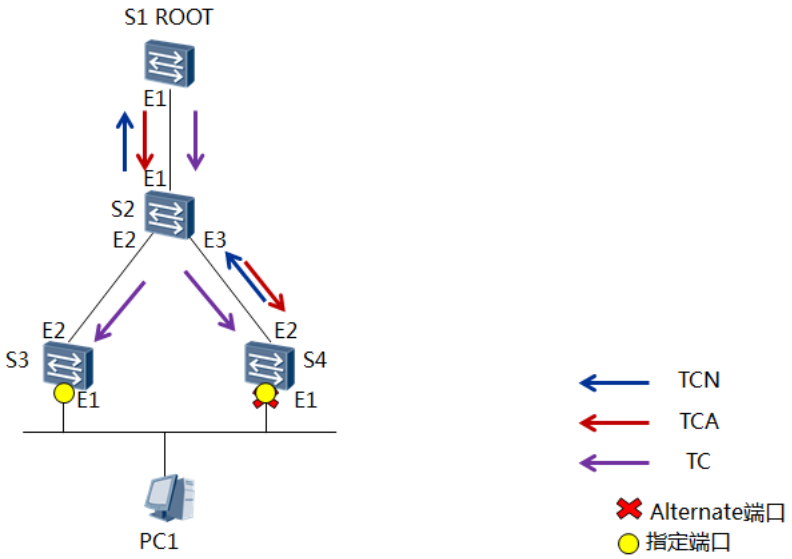
符合我们生成优先级最高的生成树的要求。

通过以上两次实验，我们验证了生成树机制的正确性。通过交换 config 信息，可以生成一个开销最小的生成树，并且优先级最高，以上的两个生成树的开销最小,而且满足优先级最高,每个节点都得到了一个开销小,优先级高的路径，我们的生成树算法实现正确

五、思考题

1.调研说明标准生成树协议中，如何处理网络拓扑变动的情况：当节点加入时？当节点离开时？

节点加入或者离开拓扑结构时候，都要重新进行 STP 计算，更新节点状态和端口的配置信息，如果生成树拓扑变化，发送数据的路径也会发生变化。拓扑变化时，应该及时更新 MAC 地址表项。拓扑变化时候，发生改变的交换机通过 TCN BPDU 报文(Topology Change Notification)告知根桥生成树拓扑里发生了变化，只有指定端口会处理 TCN BPDU。上游设备收到 TCN BPDU，回复 TCA 置位的配置 BPDU，用于终止下游设备发送 TCN BPDU。上游继续向根桥方向发送 TCN BPDU，直到根桥收到后。根桥回复 TCA+TC，TC 全网泛洪，用于清除交换机上的 mac 地址表。TC 的作用为：避免旧的 mac 表造成数据的转发错误。



- TCN 消息处理过程：
  - 在网络拓扑发生变化后，有端口转为转发状态的下游设备会不间断地向上游设备发送 TCN BPDU 报文。
  - 上游设备收到下游设备发来的 TCN BPDU 报文后，只有指定端口处理 TCN BPDU 报文。其它端口也有可能收到 TCN BPDU 报文，但不会处理。
  - 上游设备会把配置 BPDU 报文中的 Flags 的 TCA 位设置 1，然后发送给下游设备，告知下游设备停止发送 TCN BPDU 报文。
  - 上游设备复制一份 TCN BPDU 报文，向根桥方向发送。
  - 重复步骤 1、2、3、4，直到根桥收到 TCN BPDU 报文。
  - 根桥收到 TCN BPDU 后，会将下一个要发送的配置 BPDU 中的 TCA 位置位，作为对收到的 TCN 的确认，还会将该配置 BPDU 报文中的 Flags 的 TC 位置 1，用于通知所有网桥拓扑发生了变化。
  - 根桥在之后的 max age+forwarding delay 时间内，将发送 BPDU 中的 TC 置位的报文，收到该配置 BDPU 的网桥，会将自身 MAC 地址老化时间缩短为 forwarding delay。

2. 调研说明标准生成树协议是如何在构建生成树过程中保持网络连通的
提示：用不同的状态来标记每个端口，不同状态下允许不同的功能（Blocking, Listening, Learning, Forwarding 等）

端口状态	发送/接收配置BPDU	MAC地址学习	转发数据	作用说明
Disable	否/否	否	否	端口状态为down
Blocking	否/是	否	否	阻塞端口的最终状态
Listening	是/是	否	否	过渡状态，选举根桥、确定端口角色
Learning	是/是	是	否	过渡状态，构建MAC地址表
Forwarding	是/是	是	是	只有根端口和指定端口才能进入Forwarding状态

在构建生成树的过程中，STP 会将部分冗余链路强制转化为阻塞状态，其余

则链路处于转发状态。当处于转发状态的链路不可用时，STP 可以重新配置网络，集合合适的备用链路，恢复部分冗余链路的转发状态来确保网络的连通性。

3.实验中的生成树机制效率较低，调研说明快速生成树机制的原理

RSTP 概述：

快速生成树协议 RSTP（Rapid Spanning Tree Protocol）在 STP 基础上实现了快速收敛，并增加了边缘端口的概念及保护功能。

RSTP 在 STP 基础上新增加了 2 种端口角色：、Backup 端口和边缘端口。通过端口角色的增补，简化了生成树协议的理解及部署。

Backup 端口：由于学习到自己发送的配置 BPDU 报文而阻塞的端口，指定端口的备份，提供了另外一条从根节点到叶节点的备份通路。

边缘端口：如果端口位于整个交换区域边缘，不与任何交换设备连接，这种端口叫做边缘端口。边缘端口一般与用户终端设备直接连接。

边缘端口的特点

1. 边缘端口会节省 30S 的延时，端口 UP 后会立即进入转发状态。
2. 边缘端口的 UP/DOWN 不会触发拓扑改变。
3. 边缘端口收的 TC 置为的配置 BPDU 报文不会将 MAC 地址的老化时间设置为 15s。
4. 边缘端口如果收到配置的 BPDU 报文会马上变为一个普通端 ‘’ 口，进行 STP 的收敛
5. 边缘端口也会发送配置 BPDU 报文。
6. PA 协商不会阻塞边缘端口。

RSTP 的端口状态

RSTP 的端口状态在 STP 的基础上进行了改进。由原来的五种缩减为三种。

端口状态	说明
Forwarding（转发）	在这种状态下，端口既转发用户流量又处理BPDU报文。
Learning（学习）	这是一种过渡状态。在Learning下，交换设备会根据收到的用户流量，构建MAC地址表，但不转发用户流量，所以叫做学习状态。Learning状态的端口处理BPDU报文，不转发用户流量。
Discarding(丢弃)	Discarding状态的端口只接收BPDU报文。

BPDU

RSTP 中只有一种 BPDU 包，称为 RST BPDU。对配置 BPDU 的 flags 字段进行了填充，类型值做了改变，如下：

- Type： RST BPDU 包此字段类型是 2
- Flags 字段,RST 填充了 STP BPDU Flags 中的保留字段，填充了如下信息：
  1. Aggrement: 确认标识位，用于 RSTP 中定义的 Proposal/Aggrement 机制，对于 Proposal 报文的确认
  2. Forwarding: 转发状态标识位，1: 表示发送该 BPDU 报文的端口处于 Forwarding 状态
  3. Learning： 学习状态标识位，1：表示发送该 BPDU 报文的端口处于 Learning 状态

4. Port role: 端口角色标识位，00：未知角色，01：Alternat/Backup，10：根端口，11：指定端口
5. Proposal：1：表示是P/A机制中的Proposal报文

RSTP拓扑变化

触发RSTP拓扑变化的标准：非边缘端口迁移到Forwarding状态。

触发发拓扑变化的处理流程：

1. 本设备所有非边缘指定端口启动TC Whiled定时器，定时器为Hello定时器两倍。这个时间段内，清空状态发生变化的端口上学习到的MAC地址。同时，由这些端口向外发送TC:1的RST BPDU。TC While超时，则停止发送RST BPDU
2. 其他交换设备接收到TC为1的RST BPDU报文后，清空所有端口学习到的MAC地址，除了收到RST BPDU报文的位置。同时，为自己所有非边缘指定端口和根端口启动TC While定时器，重复上述过程

RSTP与STP配置BPDU处理上的变化

1. 拓扑稳定后，RSTP中所有非根桥设备仍然按照Hello定时器定期发送配置BPDU
2. 更短的BPDU超时计时。RSTP中，一个端口连续3倍Hello定时器时间内未收到上游设备发送的RST BPDU，则认为此邻居来连接失败。STP中，需要先等待Max AGE
3. 处理次等BPDU。在STP中，指定端口收到次优BPDU会立即将更优的BPDU发送出去，但对非指定端口不做如此处理。RSTP中，会对所有端口如此处理

RSTP P/A收敛机制

RSTP中，实现更快速的拓扑收敛，主要采用了Proposal/Aggrement机制。STP协议中，一个端口选举成为指定端口，至少要等待两个Forward Delay才会迁移到Forwarding状态。

RSTP中，一个端口成为指定端口后，会先进入Discarding状态，在通过P/A机制快速进入Forwarding状态。

P/A机制工作原理：

1. proposing: 当一个指定端口处于Discarding或Learning状态时，proposing变量置位，并向下游设备传递flags字段Proposal为1的RST BPDU报文，请求切换到Forwarding状态
2. proposed：当对端的根端口收到Proposal为1的RST BPDU，将此端口的proposed变量置位
3. sync: 当根端口的proposed变量置位后，会依次为本桥其他端口是sync变量置位，使所有非边缘端口都进入Discarding状态，准备重新同步
4. syncd：当端口进入到Discarding状态后，会将自己的synced的变量置位，包括本桥上的所有其他端口，包括Alternat、Backup和边缘端口。根端口会监视其他端口的sync位，当所有端口完成置位，将自己syncd置位，表示本桥已完成同步，向上游设备传回Aggreement标识为1的RST BPDU
5. agreed：当指定端口收到根端口发送的Aggreement RST BPDU，指定端口的agreed被置位。agreed被置位后，立即进入Forwarding状态。

其他收敛机制：

1. 根端口快速切换机制。如果一个根端口失效，那么网络中最优的Alternat端口立即成为根端口，并进入Forwarding状态

2. 边缘端口的引入。如果一个端口设置为边缘端口，不接收不处理 BPDU 报文，不参与 RSTP 计算，可以由 Disable 状态直接进入 Forwarding 状态。边缘端口一旦收到 BPDU 包后，会成为普通端口

RSTP 保护功能

- BPDU 保护： 边缘端口收到 RST BPDU 包，会被置为 error-down，通知网管系统
- 防 TC-BPDU 报文攻击保护：在单位时间内交换设备处理拓扑变化报文的次数
- Root 保护： 启用 root 保护功能的指定端口，其端口角色只能保持为指定端口。如果收到更优的 RST BPDU 报文时，进入 Discarding 状态
- 环路保护:如果根端口或 Alternat 端口长时间收不到来自上游的 RST BPDU，向网关发送通知。根端口会进入 Discarding 状态。阻塞端口一致阻塞，指导根端口收到 RST BPDU 才恢复正常

RSTP 与 STP 相比，主要有以下几点优势：

- 如果旧的根端口已经进入阻塞状态，而且新根端口连接的对端交换机的指定端口处于 Forwarding 状态,在新拓扑结构中的根端口可以立刻进入转发状态;
- 网络边缘的端口，即直接与终端相连，而不是和其它网桥相连的端口可以直接进入转发状态，不需要任何延时;
- 增加了网桥之间的协商机制—Proposal/Agreement。指定端口可以通过与相连的网桥进行一次握手，快速进入转发状态。