Stp 实验设计报告

中国科学院大学

页

2022 年 4 月 16 日

一、生成树机制实验

1. 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。

2. 实验流程

（1）实现生成树运行机制: 主要是补充完成 config 处理函数函数：

```
static void stp_handle_config_packet(stp_t *stp, stp_port_t *p,
        struct stp_config *config)
```

添加 swich 实验中的 broadcast.c、mac.c、mac.h 和 list.h 等文件到本实验文件夹中。

（2）根据添加的文件完善 makefile，编译 make 生成 stp，在给定 4 节点拓扑下运行生成树机制。运行 four_node_ring.py 拓扑，4 个节点分别运行 stp 程序，将输出重定向到 b*-output.txt 文件，以 b1 为例：

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

等待一段时间(4 个节点大概 30 秒钟)后，执行如下命令：

(b?/root)# pkill -SIGTERM stp

该命令强制所有 stp 程序输出最终状态并退出

执行 dump_output.sh 脚本，输出个 4 个节点的状态

./dump_output.sh 4

将上述命令补充到 four_node_ring.py 文件中以便快速执行

```
    for idx in range(4):
        name = 'b' + str(idx+1)
        node = net.get(name)
```

```
        clearIP(node)
        node.cmd('./scripts/disable_offloading.sh')
        node.cmd('./scripts/disable_ipv6.sh')

        # set mac address for each interface
        for port in range(len(node.intfList())):
            intf = '%s-eth%d' % (name, port)
            mac = '00:00:00:00:0%d:0%d' % (idx+1, port+1)

            node.setMAC(mac, intf = intf)

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

    net.start()
    sleep(32)
    for idx in range(4):
        name = 'b' + str(idx+1)
        node = net.get(name)
        node.cmd('pkill -SIGTERM stp')

    sleep(1)
    os.system('./dump_output.sh 4')
    #raw_input()
    #CLI(net)
    net.stop()
```

（3） 建立 7 节点拓扑结构：

```
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, b5)
        self.addLink(b4, b3)
        self.addLink(b4, b7)
        self.addLink(b5, b3)
        self.addLink(b5, b6)
        self.addLink(b6, b3)
        self.addLink(b6, b7)
        self.addLink(b7, b3)
```

利用上述拓扑结构仿照 four_node_ring.py 得到 seven_node_ring.py，执行脚本。

（4） 撰写实验报告，完成思考题。

3. 实验设计

在 swich 广播实验已经解决了通信等问题，本实验的代码框架中，初始化后，每个节点的 stp 程序可分为三个线程。其中线程 1 每隔一个 hello time（2s）遍历节点的每个指定端口，向端口所处网段发送记录的 config 包。线程 2 监听本节点收到的包，对于每个端口收到的包，若其为 stp 协议 config 包，则需要利用 stp_handle_config_packet 函数处理该包，具体的处理请见后文。线程 3 每隔一段时间扫描 mac-端口映射表 map，老化删除过期的条目。

本次实验设计需要解决的问题主要是实现 stp_handle_config_packet 函数。首先，理解该函数需要维护的数据结构是有必要的：

端口相关数据结构（stp.h）：

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

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

```
    int path_cost;              // 端口所在网段的开销

    u64 designated_root;        // 自己认为的根节点
    u64 designated_switch;      // 本网段到根节点的上一跳节点 ID
    int designated_port;        // 本网段到根节点的上一跳端口
    int designated_cost;        // 本网段到根节点的路径开销
};
```

节点数据结构（stp.h）：

```
struct stp {
    u64 switch_id;

    u64 designated_root;      // 自己认为的根节点
    int root_path_cost;       // 到根节点的路径开销
                              //节点到根节点的路径开销等于根端口所在网段到根
                             //节点的路径开销与根端口所在网段的通过开销之和
    stp_port_t *root_port;      // lowest cost port to root

    long long int last_tick;    // switch timers

    stp_timer_t hello_timer;    // hello timer（2s）

    // ports
    int nports;                   // 节点拥有端口数
    stp_port_t ports[STP_MAX_PORTS];

    pthread_mutex_t lock;
    pthread_t timer_thread;
};
```

stp_handle_config_packet 函数将处理收到的 config 包，提取需要的 4 个字段：根节点 ID，本网段到根节点的开销，本网段的指定端口所在节点的 ID，本网段指定端口的 ID。本次设计利用 config_to_port 函数记录这些信息，函数提取 config 报文中的信息到 p 中：

```
static void config_to_port(struct stp_config *config, stp_port_t *p) {
    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);
}
```

然后根据收到的信息与本端口和本端口对应节点的 config 进行比较优先级，来更新本端口和本节点的 config。同网段两个端口 config 的优先级比较如下：根节点 ID 小的优先级

高；若相同，到根节点开销小的优先级高；若相同，到根节点上一跳节点（即该网段的指定端口所在节点）ID 小的优先级高；若相同，到根节点上一跳端口（即该网段的指定端口）ID 小的优先级高。具体实现如 compare_proi 函数，函数返回值为 1 则 p 的优先级高于 q 的优先级：

```
int compare_proi(stp_port_t *p, stp_port_t *q)
{
    if (q->designated_root != p->designated_root)
    {
        return q->designated_root > p->designated_root;
    }else if(q->designated_cost != p->designated_cost)
    {
        return q->designated_cost > p->designated_cost;
    }else if(q->designated_switch != p->designated_switch)
    {
        return q->designated_switch > p->designated_switch;
    }else if(q->designated_port != p->designated_port)
    {
        return q->designated_port > p->designated_port;
    }
    else
    {
        return 0;
    }
}
```

比较本端口和发送端口的优先级后倘若对方端口优先级更高则需要进行以下处理以下操作：

- 更新本端口的 config
- 更新本端口对应的本节点的 config
- 更新本节点其他端口的 config
- 若本节点从根节点变为非根节点，则停止本节点的计时器，并将 config 通过指定节点发出

首先寻找根端口。遍历所有端口，根端口**应满足：该端口是非指定端口，该端口的优先级要高于所有剩余非指定端口。**

```
        stp_port_t *root_port = NULL;
        stp_port_t *temp = NULL;
        for (int i = 0; i < stp->nports; i++)
        {
            root_port = &stp->ports[i];
```

```
            if(!stp_port_is_designated(root_port))
            {
                for (int j = 0; j < stp->nports; j++)
                {
                    temp = &stp->ports[j];
                    if(!stp_port_is_designated(temp) && root_port!=temp)
                         if(!compare_proi(root_port,temp))
                        {
                            root_port = temp;
                             break;
                        }
                }
                if(root_port == &stp->ports[i])
                     break;
                else
                     root_port = NULL;
            }
        }
```

如果不存在根端口，则该节点为根节点，否则，选择通过 root_port 连接到根节点，更新节点状态。

```
        if(root_port)
        {
           stp->root_port = root_port;
           stp->designated_root = root_port->designated_root;
           stp->root_path_cost  = root_port->designated_cost +
root_port->path_cost;
        }else
        {
           stp->root_port = root_port;
           stp->designated_root = stp->switch_id;
           stp->root_path_cost = 0;
        }
```

接下来更新节点中其他端口的状态，检查所有的网段，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。对于一个端口，先假设其可以成为指定端口，然后构造其现在的 config，若构造的 config 优先级高于网段内持有的最高优先级，则它可以成为指定断点。

```
                build->designated_root = stp->designated_root;
                build->designated_switch = stp->switch_id;
```

```
                build->designated_port = temp->port_id;
                build->designated_cost = stp->root_path_cost;
                if (compare_proi(build, temp))
                {
                    temp->designated_switch = stp->switch_id;
                    temp->designated_port = temp->port_id;
                }
```

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：

```
        for (int i = 0; i < stp->nports; i++)
        {
            stp_port_t *temp = &stp->ports[i];
            if (stp_port_is_designated(temp))
            {
                temp->designated_root = stp->designated_root;
                temp->designated_cost = stp->root_path_cost;
            }
        }
```

该节点原本是根节点，并且此时变成了非根节点，则需要停止 hello 计时器

```
        if (root_before && !stp_is_root_switch(stp))
            stp_stop_timer(&stp->hello_timer);
```

利用上述逻辑我们将能是构造优先级最高的唯一的生成树。生成树机制收敛后，每个网段内所有端口存储的 config 包的信息都相同，每个节点认为的根节点都相同。只有根端口和指定端口可以转发数据包。另外，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

4. 实验结果分析

（1）在给定 four_node_ring.py 的 4 节点拓扑下运行生成树机制，在进行生成树算法前，构建的网络拓扑结构如图 1：

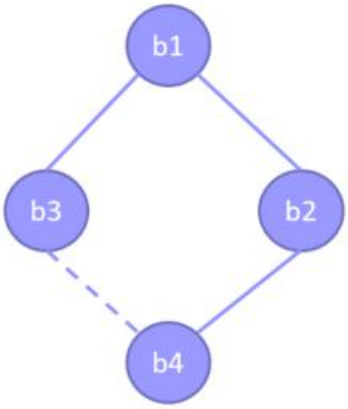

图 1

运行生成树算法后打印信息如下：

```
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.
```

其对应的拓扑结构如图 2，节点拓扑形成了一棵生成树：

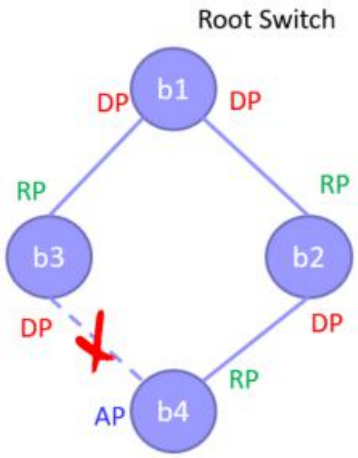

图 2

（2）在 seven_node_ring.py 的 7 节点拓扑下运行生成树机制，在进行生成树算法前，构建的网络拓扑结构如图 3：

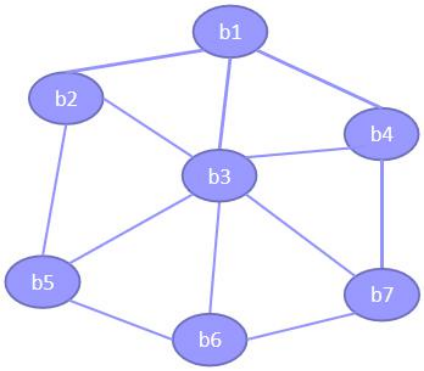

图 3

其运行生成树算法后打印信息如下：

```
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.
INFO: port id: 05, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 05, ->cost: 1.
INFO: port id: 06, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 06, ->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: 0201, ->port: 03, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 04, ->cost: 1.
INFO: port id: 03, role: DESIGNATED.
INFO:   designated ->root: 0101, ->switch: 0501, ->port: 03, ->cost: 2.

NODE b6 dumps:
INFO: non-root switch, designated root: 0101, root path cost: 2.
INFO: port id: 01, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0501, ->port: 03, ->cost: 2.
INFO: port id: 02, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 05, ->cost: 1.
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: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0401, ->port: 03, ->cost: 1.
INFO: port id: 02, role: ALTERNATE.
INFO:   designated ->root: 0101, ->switch: 0601, ->port: 03, ->cost: 2.
INFO: port id: 03, role: ROOT.
INFO:   designated ->root: 0101, ->switch: 0301, ->port: 06, ->cost: 1.
```

其对应的拓扑结构如图 4，节点拓扑形成了一棵生成树：

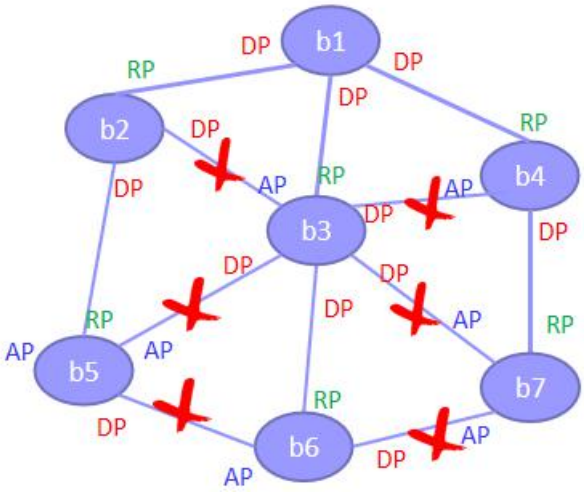

图 4

5. 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？

STP 协议包格式称为 BPDU（Bridge Protocol Data Unit）。配置 BPDU 是一种心跳报文，只要端口使能 STP，则配置 BPDU 就会按照 Hello Time 定时器规定的时间间隔从指定端口发出，TCN BPDU 是在设备检测到网络拓扑发生变化时才发出。，根节点继续周期性发送配置 BPDU 信息，非指定端口会不断的侦听对端发送的 BPDU，如果超过一定时间内没有收到 配置 BPDU 信息，非指定端口认为网络发生变化，网络将会重新进行收敛计算。假设拓扑发生了变动，将按照下面的方式重新进行 STP 计算，生成新的树形拓扑.

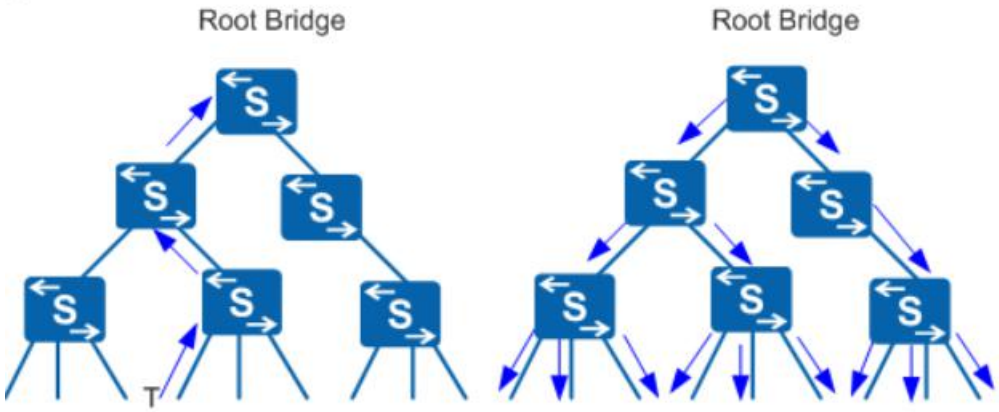

图 5

1. T 点接口发生变更后，下游设备会不间断地向上游设备发送 TCN BPDU 报文。
2. 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。

3. 上游设备会把配置 BPDU 报文中的 Flags 的 TC 位和 TCA 位同时置 1，然后发送给下游设备，告知下游设备停止发送 TCN BPDU 报文。
4. 上游设备复制一份 TCN BPDU 报文，向根桥方向发送。
5. 重复步骤 1、2、3、4，直到根桥收到 TCN BPDU 报文。
6. 根桥把配置 BPDU 报文中 Flags 的 TC 位和 TCA 位同时置 1 后发送，TC 位置 1 是为了通知下游设备直接删除桥 MAC 地址表项，TCA 位置 1 是为了通知下游设备停止发送 TCN BPDU 报文。

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）。

标准生成树协议中, 端口有以下几种状态:

- ✧ Blocking（阻塞状态）：二层端口为非指定端口，也不会参与数据帧的转发。
- ✧ Listening（侦听状态）：生成树会根据交换机所接收到的 BPDU 而判断出了这个端口应该参与数据帧的转发。
- ✧ Learning (学习状态)：这个二层端口准备参与数据帧的转发，并开始填写 MAC 表。在默认情况下，端口会在这种状态下停留 15 秒钟时间。
- ✧ Forwarding（转发状态）：这个二层端口已经成为了活动拓扑的一个组成部分，它会转发数据帧，并同时收发 BPDU。
- ✧ Disabled（禁用状态）：这个二层端口不会参与生成树，也不会转发数据帧

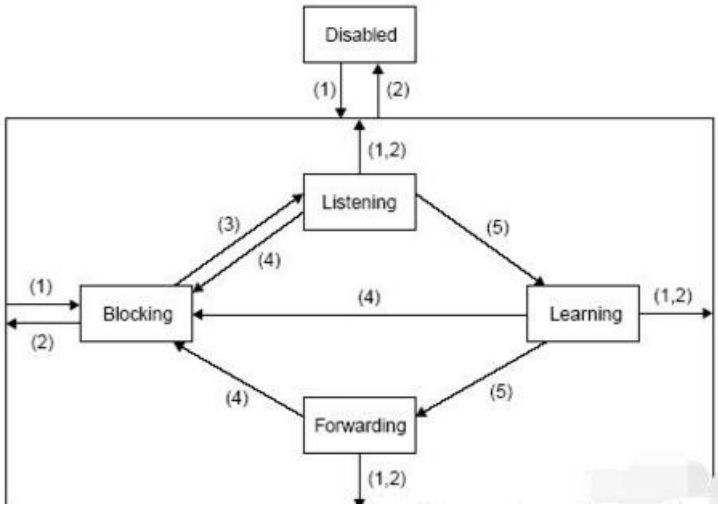

图 6

在构建生成树的过程中，STP 会将工作链路处于转发状态，转发状态是唯一的可以接收和发送用户数据的状态，其他部分冗余链路强制转化为阻塞状态，。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重新构成生成树，这样既避免了存环在路而导致的广播风暴，同时又确保网络的连通性。

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。

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 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：

- ✧ 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。

STP 缺点：1.收敛慢，learning 到 forwarding 需要 15s 学习 mac 地址；Listening 到 learning 需要 15s 加速老化时间。无论上述用了多少秒，最大等待时间总是 15s，同时发送 TCN-BPDU 的时间过长。 2.链路利用率低

在 RSTP 快速生成树中边缘端口不会进行端口角色计算，直接变成 forwarding 状态，RSTP 具有 Proposal/Agreement 机制。对于 STP 为了避免环路，必须等待足够长的时间，使全网的端口状态全部确定，也就是说必须要等待至少一个 Forward Delay 所有端口才能进行转发。Proposal/Agreement 过程示意如下：

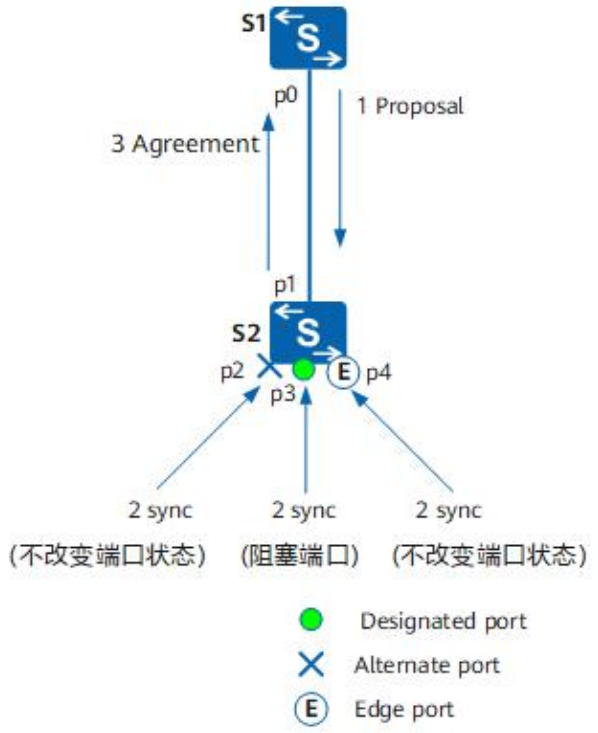

图 7

新链路连接成功后，P/A 机制协商过程如下：

1. p0 和 p1 两个端口马上都先成为指定端口，发送 RST BPDU。
2. S2 的 p1 口收到更优的 RST BPDU，马上意识到自己将成为根端口，而不是指定端口，停止发送 RST BPDU。
3. S1 的 p0 进入 Discarding 状态，于是发送的 RST BPDU 中把 Proposal 和 Agreement 置 1。
4. S2 收到根桥发送来的携带 Proposal 的 RST BPDU，开始将自己的所有端口进入 sync 变量置位（即同步变量：临时阻塞除边缘端口外的其他端口）。
5. p2 已经阻塞，状态不变；p4 是边缘端口，不参与运算；所以只需要阻塞非边缘指定端口 p3。
6. 各端口的 sync 变量置位后，p2、p3 进入 Discarding 状态，p1 进入 Forwarding 状态并向 S1 返回 Agreement 位置位的回应 RST BPDU。
7. 当 S1 判断出这是对刚刚发出的 Proposal 的回应，于是端口 p0 马上进入 Forwarding 状态。
8. 下游设备继续执行 P/A 协商过程。

RSTP 通过阻塞自己的非根端口来保证不会出现环路，使用 P/A 机制加快了上游端口转到 Forwarding 状态的速度。相比较 STP 模式时接口需要等待 2*Forward Delay 时间才能进入 Forwarding 状态，RSTP 模式时的根端口快速切换机制使接口直接切换成 Forwarding 状态，减少业务流量丢包。

RSTP 定义了两种新的端口角色：备份端口（Backup Port）和预备端口（Alternate Port）。根据 STP 的不足，RSTP 新增加了两种端口角色，并且把端口属性充分地按照状态和角色解耦，使得可以更加精确地描述端口，从而使得协议状态更加简便，同时也加快了拓扑收敛。通过端口角色的增补，简化了生成树协议的理解及部署。

在 RSTP 里面，如果某一个指定端口位于整个网络的边缘，即不再与其他交换设备连接，而是直接与终端设备直连，这种端口叫做边缘端口。边缘端口不参与 RSTP 运算，可以由 Disable 直接转到 Forwarding 状态，且不经历时延，就像在端口上将 STP 禁用。但是一旦边缘端口收到配置 BPDU，就丧失了边缘端口属性，成为普通 STP 端口，并重新进行生成树计算。

参考资料：

【1】 STP 的拓扑变化机制。
【2】 交换机网络的生成树协议简述。

【3】 STP 端口状态。
【4】 STP 算法研究。
【5】 STP 协议原理与配置整——STP 的不足。
【6】 RSTP（快速生成树）。
【7】 RSTP 如何实现快速收敛。