Linux SRv6 实战：VPN、流量工程和服务链

（第一篇）

李嘉明 思科系统工程师

苏远超 思科首席工程师

钟　庆 思科系统工程师

摘要：本文基于 Linux SRv6 功能，结合 Mininet、Quagga、Python 等工具，验证 SRv6 的一系列功能，包括 VPN、流量工程、服务链等。

本文所有代码见: https://github.com/ljm625/srv6_Sandbox

一、SRv6 简介

Segment Routing（以下简称 SR）指由思科发明，并主要由 IETF SPRING（Source Packet Routing In Networking）工作组进行标准化的新一代网络传送技术。SR 基于源路由并且只在网络边缘维持状态，这使得 SR 非常适合于超大规模 SDN 部署，在极大简化网络的同时，SR 也为网络提供了高度的可编程能力以及端到端的流量工程能力。因此在出现仅短短五年后，SR 已经成为业界共识，是新一代网络尤其是 5G 网络的事实 SDN 架构标准。

SR 数据平面有两种实现方式，一种是 SR MPLS，重用了 MPLS 数据平面；另一种是 SRv6，使用 IPv6 数据平面。SR 架构可以运行在这两种数据平面上，这是自 SR 提出第一天起就确立的原则。

SRv6 采用 IPv6 标准中定义的路由扩展报头(Routing Extension Header)承载新定义的 SRH（Segment Routing Header）扩展路由报头，SRH 类型号定义为 4。在 SRH 中包含了 Segment 列表。SRv6 Segment 形式上是一个 128 位的 IPv6 的地址，但其实此 Segment 由 Locator(定位器)和 Function(指令)构成(还可以含有”参数”信息,本文先略过)，Locator 用于 IPv6 路由，Function 用于指定节点需要对数据包施加的各种 SRv6 操作，实现网络的可编程性。

和 SR MPLS 不一样，在数据包的转发过程中 SRv6 通常不会弹出 Segment，而是通过 SRH 中的 Segment Left(剩余 Segment，是个不小于 0 的数值)字段作为指针，指向活动 Segment，类似于 SR MPLS 中的顶层标签。每经过一个 SRv6 端节点，

Segment Left 减 1，更新 IPv6 报头的目的地址为 Segment 列表中当前 Segment Left 对应的 Segment，并遵循常规的 IPv6 路由把数据包转发出去。

需要强调的是， 如果网络中有节点只支持常规的 IPv6 而不支持 SRv6，当此节点收到 SRv6 数据包时, 按照 IPv6 RFC 的规定，由于数据包目的地址不是节点自身网段地址, 此节点不处理扩展报头，而只是单纯地根据数据包目的地址进行 IPv6 转发。这意味着，SRv6 可以与现有的 IPv6 网络无缝互操作，换句话说，SRv6 可以在 IPv6 网络上实现增量部署，无须替换现网所有设备。

以下是本文用到的 SRv6 操作简介：

End : 该操作要求 Segment Left 不为 0（不是最后一跳），会将 Segment Left 减 1，并更新 IPv6 数据包的目的地址为下一个 Segment，这是最常见的 SRv6 操作。相当于 SR MPLS 中的 Prefix-SID。

End.X：该操作和 End 操作基本一致，区别是可以将处理过的数据包发送到指定的下一跳地址。相当于 SR MPLS 中的 Adj-SID。

End.DX4：该操作要求 Segment Left 为 0 且数据包内封装了 IPv4 数据包，会去掉外层的 IPv6 报头，并将内部的 IPv4 数据包转发给指定的下一跳地址。相当于 VPNv4 Per-CE 标签。

End.DX6：该操作要求 Segment Left 为 0 且数据包内封装了 IPv6 数据包，会去掉外层的 IPv6 报头，并将内部的 IPv6 数据包转发给指定的下一跳地址。相当于 VPNv6 Per-CE 标签。

End.B6：该操作会在已有的 SRH 的基础上，插入一个新的 SRH，并可以定义新的 Segment 列表，数据包将首先按照插入的新的 SRH 进行转发。相当于 Binding-SID。

End.B6.Encaps：该操作和 End.B6 基本一致，区别是该操作将在数据包外层新增一个新的 IPv6 报头和 SRH，而不是仅仅添加一个 SRH 的路由报头。

T.Encaps：该操作在中转节点（即数据包经过的支持 SRv6 的路由器，但节点本身不在 Segment 列表中）上执行，会在数据包外层新加一个 IPv6 报头以及 SRH 报头，并可以定义新的 Segment 列表，数据包将首先按照新 IPv6 报头中的 SRH 进行转发。

二、Why Linux SRv6?

1. 试验环境容易建立

   Linux 内核从 4.10 版本(2017 年 2 月)开始就支持 SRv6，距今已经有差不多 2 年的时间，功能已经比较成熟。事实上本文的所有测试，都是在一台主机上借助 Mininet 完成。

   另一方面，虽然思科在业界率先在网络设备上支持了 SRv6，但总体而言，目前业界网络设备对 SRv6 的支持程度还比较有限，特别是对于流量工程和服务链等高级功能的支持，因此当前用网络设备不能完全体现出 SRv6“极简+编程”

的革命性创新。但我们相信，随着 SRv6 成为业界共识，网络设备对 SRv6 的支持也会越来越好。

2. 支持丰富的 SRv6 操作

目前 Linux 对 SRv6 的支持情况如下表所示：

Name	Description	Release
End	Endpoint function	4.10 (February 2017), srext
End.X	Endpoint function with Layer-3 cross-connect	4.10 (February 2017), srext
End.T	Endpoint function with specific IPv6 table lookup	4.14 (November 2017)
End.DX2	Endpoint with decapsulation and Layer-2 cross-connect	4.14 (November 2017), srext
End.DX6	Endpoint with decapsulation and IPv6 cross-connect	4.14 (November 2017), srext
End.DX4	Endpoint with decapsulation and IPv4 cross-connect	4.14 (November 2017), srext
End.DT6	Endpoint with decapsulation and IPv6 table lookup	4.14 (November 2017)
End.DT4	Endpoint with decapsulation and IPv4 table lookup	In development
End.B6	Endpoint bound to an SRv6 policy	4.14 (November 2017)
End.B6.Encaps	Endpoint bound to an SRv6 encapsulation Policy	4.14 (November 2017)
End.BM	Endpoint bound to an SR-MPLS Policy	In development
End.S	Endpoint in search of a target in table T	In development
End.AD	Endpoint to SR-unaware APP via dynamic proxy	srext
End.AM	Endpoint to SR-unaware APP via masquerading	srext

Name	Description	Release
T.Insert	Transit behavior with insertion of an SRv6 Policy	4.10 (February 2017)
T.Encaps	Transit behavior with encapsulation in an SRv6 policy	4.10 (February 2017)
T.Encaps.L2	T.Encaps behavior of the received L2 frame	4.14 (November 2017)

表 1：Linux 支持 SRv6 情况

可以看到 Linux 已经支持大部分 SRv6 功能，但部分功能需要使用 srext 这个内核扩展模块来实现。本文中我们将只使用 Linux 内核中原生支持的功能进行验证，srext 支持的功能将在下一篇文章中进行讲述。

3. 适用于虚拟化/叠加网络环境

在数据中心/云端广泛采用 Linux+NFV 技术提供防火墙、入侵检测（IDS）、负载均衡等增值网络业务及服务链(Service Chaining)功能，Linux SRv6 可以在其中大显身手。

另一方面，不少用户在数据中心/云端采用主机叠加网络(Host Overlay)来为租户服务，但如何在 Host Overlay 与 Underlay 间实现无缝耦合、保证 SLA 一直是个难题。Linux SRv6 可以完美整合 Overlay 和 Underlay，因为无论是 Overlay 还是 Underlay，本质上都是对应着不同的 SRv6 Segment（操作）而已；如果需要提高 Linux SRv6 性能，可以优化 DPDK 或者使用 FD.IO(VPP)，其中 FD.IO 已经内置了完善的 SRv6 支持，在使用上会更为方便。

三、准备工作

验证环境基于 Ubuntu 以及 Mininet，也可以使用 Vagrant+Virtualbox 实现。

- Linux，要求内核版本高于 4.15
- 最新版 Mininet
- Quagga（在 Mininet 虚拟拓扑下，提供路由器的静态路由/OSPF/BGP 等路由协议支持）
- Python （通过脚本建立测试拓扑及初始配置）

四、安装教程

下面的安装教程基于 Ubuntu 18.04 LTS。

1. 升级内核到推荐版本(4.15.0-38)

```
apt-get update
apt-get install linux-headers-4.15.0-38 linux-headers-4.15.0-38-generic linux-image-4.15.0-38-generic linux-modules-4.15.0-38-generic linux-modules-extra-4.15.0-38-generic
```

2. 重启，检查内容是否安装完成

```
uname -a
Linux ubuntu 4.15.0-38-generic #41-Ubuntu SMP Wed Oct 10 10:59:38 UTC 2018 x86_64 x86_64 x86_64 GNU/Linux
```

3. 安装 Mininet 和 Quagga

```
apt-get install mininet gawk libreadline-dev libc-ares-dev
wget http://download.savannah.gnu.org/releases/quagga/quagga-1.2.4.tar.gz
tar -xzvf ./quagga-1.2.4.tar.gz
cd ./quagga-1.2.4
./configure --enable-vtysh --enable-user=root --enable-group=root --enable-vty-group=root
make install
```

4. 安装最新版的 iproute2

```
wget https://mirrors.edge.kernel.org/pub/linux/utils/net/iproute2/iproute2-4.9.0.tar.gz
tar -xzvf ./iproute2-4.9.0.tar.gz
cd ./iproute2-4.9.0
apt-get install bison flex
make
make install
```

5. 安装 Python 的依赖包

```
pip install mako ipaddress ipmininet --no-deps
```

6. 下载实验拓扑的配置文件

```
git clone https://github.com/ljm625/SRv6_Sandbox
```

五、验证场景

5.1 Linux SRv6 实现 VPN+流量工程

5.1.1 概述

目的：使 2 台仅支持 IPv4 的主机（主机 a 和主机 b），通过 SRv6 实现 VPN 互通，并实现流量工程。

拓扑如下图所示：

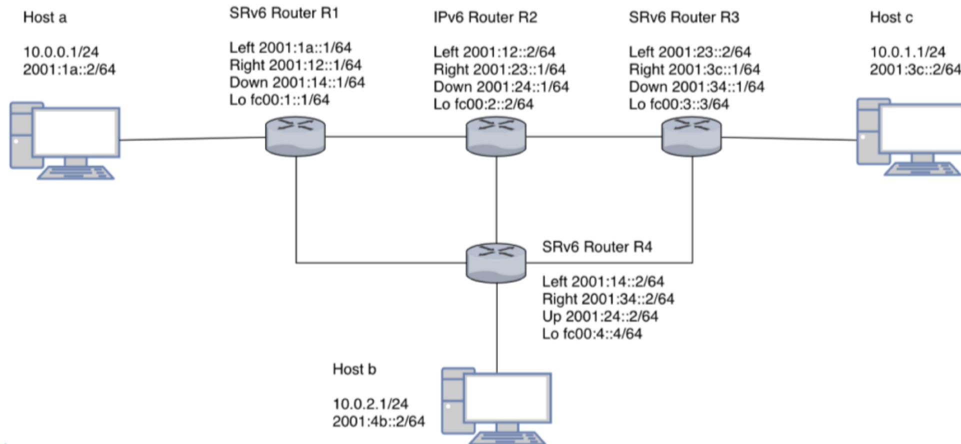

图 1：Linux SRv6 实现 VPN+流量工程拓扑

图 1 中路由器 R1、R3 和 R4 为支持 SRv6 的路由器，R2 为仅支持 IPv6 的路由器，通过开源 Quagga 实现静态路由，路由器与路由器之间仅通过 IPv6 实现互通。

在这个测试中，我们的目的是让主机 a 与主机 b 实现 IPv4 互通，并让数据包经由 R3 路由器，从而实现 VPN 及流量工程。

详细的数据包转发流程及每一跳的报头变化如下图所示，图中报文上的数字表示数据包在网络中的转发顺序。

请注意: 图中的 Segment 列表是逆序排列的，即排在列表的第一个 Segment 是路径上的最后一跳，排在列表的最末位 Segment 是路径的第一跳。

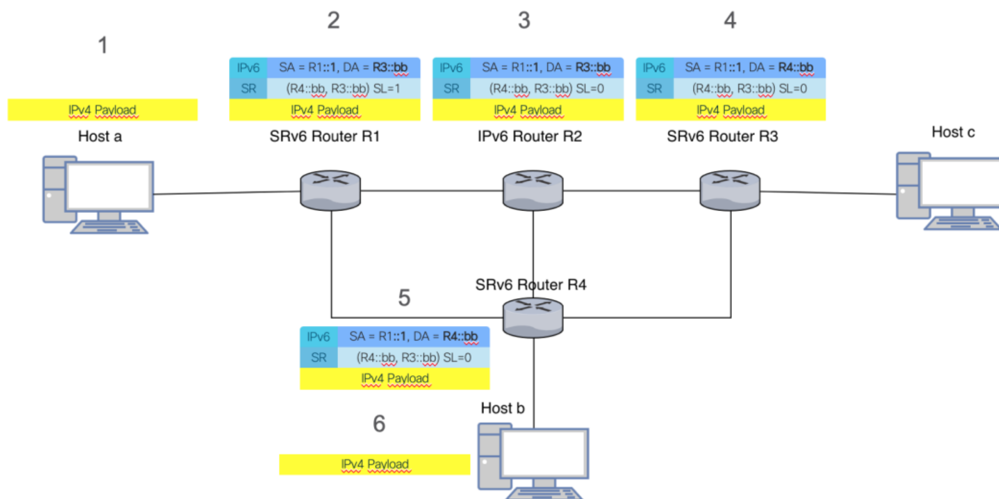

图 2：Linux SRv6 实现 VPN+流量工程-数据包转发流程

从主机 a 发出的数据包，到达支持 SRv6 的路由器 R1，R1 会根据所配置的操作对数据包进行封装，在外层加上 IPv6 以及 SRH 的报头，并进行正常的 IPv6 转发。在仅支持 IPv6 的路由器 R2，R2 根据 IPv6 报头基于目的 IPv6 地址进行转发。在

R3，R3 路由器根据 Segment 执行 End 操作，将 Segment Left 减 1，并根据 Segment 列表更新 IPv6 的目的地址，将数据包转发至下一跳 R4。在支持 SRv6 的路由器 R4，R4 根据 Segment 执行 End.DX4 操作，剥掉外层的 IPv6 报头，将内含的 IPv4 数据包发给主机 b，完成转发流程。

5.1.2 具体步骤

1. 首先启动实验拓扑

```
sudo python topo.py
```

2. 在 R1 路由器上配置 T.Encaps 操作(SRv6 流量工程)，将去往 10.0.2.0/24 的数据包，封装入 SRv6，并配置 SRH 包含的 Segment 列表为 <R4::bb,R3::bb>(逆序排列)

```
ip route add 10.0.2.0/24 encap seg6 mode encap  segs fc00:3::bb,fc00:4::bb dev r1-eth1
```

   后面的步骤会给出 Segment fc00:3::bb,fc00:4::bb fc00:4:bb 的定义。

3. 同时在 R1 上配置针对回程数据包的 End.DX4 操作，让去往主机 a 的数据包在 R1 做 IPv6 的解封装，解出 IPv4 数据包后发送给主机 a

```
ip -6 route add fc00:1::bb/128 encap seg6local action End.DX4 nh4 10.0.0.1 dev r1-eth1
```

4. 在 R3 路由器上配置 End 操作，以让 R3 在收到 R1 发来的数据包时，Segment Left 减 1，并更新 IPv6 目的地址为当前 Segment Left 指定的 Segment。

```
ip -6 route add fc00:3::bb/128 encap seg6local action End dev r3-eth2
```

5. 最后在 R4 路由器上配置 End.DX4 操作，以让 R4 收到数据包之后能够做 IPv6 解封装，并转发给指定地址。

```
ip -6 route add fc00:4::bb/128 encap seg6local action End.DX4 nh4 10.0.2.1 dev r4-eth1
```

6. 相应地在 R4 上配置 T.Encaps 操作，对 Ping 回程 IPv4 数据包进行封装。

```
ip route add 10.0.0.0/24 encap seg6 mode encap  segs fc00:1::bb dev r4-eth1
```

配置完成后，可以从主机 a(10.0.0.1) Ping 通主机 b(10.0.2.1)。

5.1.3 脚本执行及抓包验证

为了方便，我们将相关的配置都通过 Python 脚本实现了自动化。代码见 https://github.com/ljm625/srv6_Sandbox

1. R1

在 R1 上配置 SRv6。

```
"Node: r1"
root@ubuntu:~/topo/service_chain# bash -x ./config/dx4_r1.sh
+ echo 'Adding DX4 Rules for R1'
Adding DX4 Rules for R1
+ ip -6 route add fc00:1::bb/128 encap seg6local action End.DX4 nh4 10.0.0.1 dev
 r1-eth1
RTNETLINK answers: File exists
+ ip route add 10.0.2.0/24 encap seg6 mode encap segs fc00:3::bb,fc00:4::bb dev
r1-eth1
RTNETLINK answers: File exists
root@ubuntu:~/topo/service_chain#
```

图 3：R1 配置脚本

我们从主机 a(10.0.0.1) Ping 主机 b(10.0.2.1)进行测试。

然后抓包检查，可以看到从主机 a 发来的 IPv4 原始数据包。

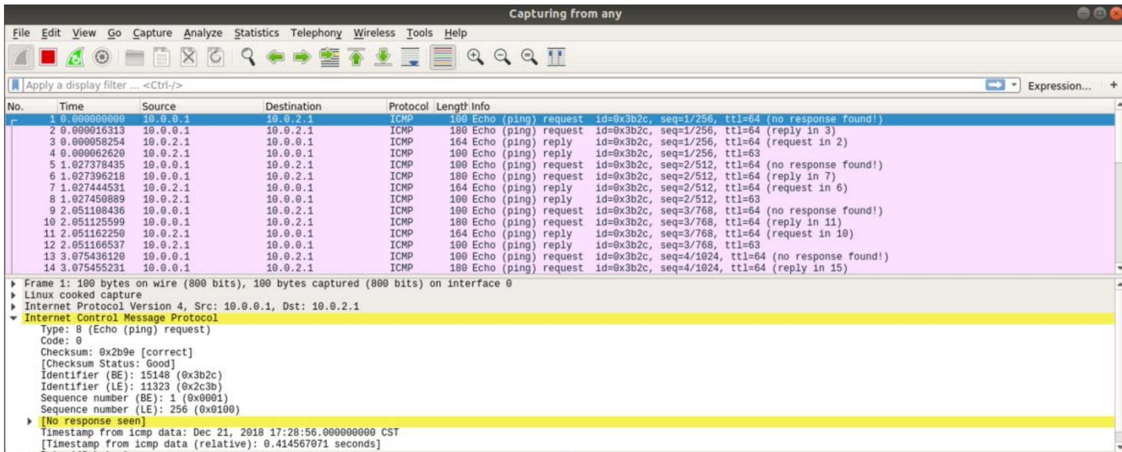

图 4：R1 上抓包-主机 a 发出的 IPv4 数据包

经过 R1 路由器后，可以看到数据包外层加了 IPv6 的报头，路由头类型(Routing Header Type)是 4（Segment Routing），里面有 2 个 Segment< fc00:4::bb ,fc00:3::bb>，Segment Left=1，所以 IPv6 目的地址设置为列表位置为 1 的地址，即 fc00:3::bb。

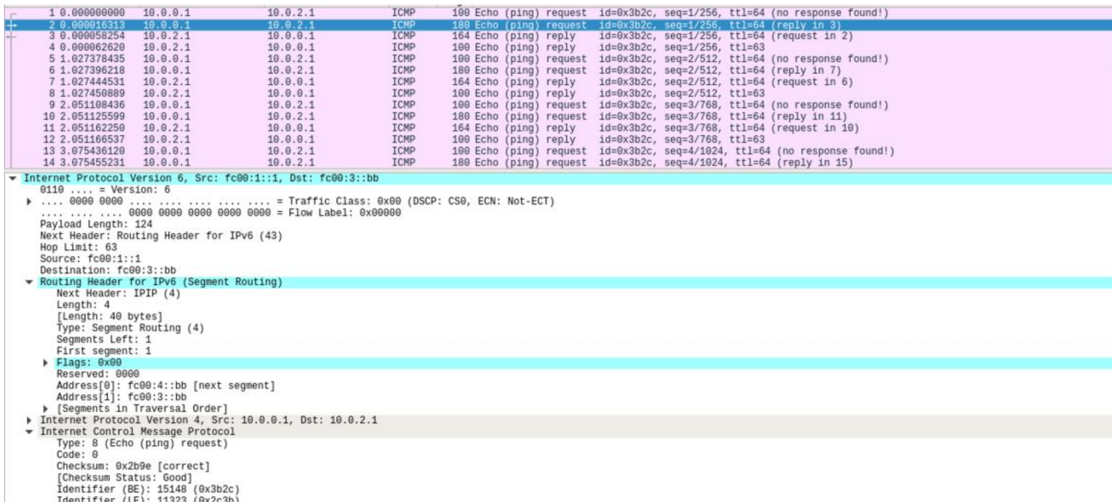

图 5：R1 上抓包-R1 生成带有 2 个 Segment 的 SRv6 数据包

2. R3

在 R3 上配置 SRv6。

图 6：R3 配置脚本

R3 从不支持 SRv6 的路由器 R2 处收到了 R1 发来的数据包，根据定义的策略会执行 End 操作，即 Segment Left 减 1，并更新 IPv6 目的地址。

```
No. Time         Source                Destination           Protocol Length Info
  1 0.000000000  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=1/256, ttl=64 (no response found!)
  2 0.000012678  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=1/256, ttl=64 (no response found!)
  3 1.006913796  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=2/512, ttl=64 (no response found!)
  4 1.006923478  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=2/512, ttl=64 (no response found!)
  5 2.031067535  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=3/768, ttl=64 (no response found!)
  6 2.031077548  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=3/768, ttl=64 (no response found!)
  7 3.055066108  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=4/1024, ttl=64 (no response found!)
  8 3.055075932  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3c70, seq=4/1024, ttl=64 (no response found!)
  9 5.135852880  fe80::4a4:3ff:fef9:…  2001:34::2            ICMPv6    88 Neighbor Solicitation for 2001:34::2 from 06:a4:03:f9:32:fa
 10 5.135865190  fe80::282d:57ff:fef…  2001:23::2            ICMPv6    88 Neighbor Solicitation for 2001:23::2 from 2a:2d:57:f7:46:70
 11 5.136002175  2001:23::2            fe80::282d:57ff:fef…  ICMPv6    80 Neighbor Advertisement 2001:23::2 (rtr, sol)
 12 5.135993864  2001:34::2            fe80::4a4:3ff:fef9:…  ICMPv6    80 Neighbor Advertisement 2001:34::2 (rtr, sol)
 13 10.254775457 fe80::c0a4:63ff:fe1…  fe80::282d:57ff:fef…  ICMPv6    88 Neighbor Solicitation for fe80::282d:57ff:fef7:4670 from c2:a4:63:1e:db:24
 14 10.254788426 fe80::f043:1ff:fe35…  fe80::4a4:3ff:fef9:…  ICMPv6    88 Neighbor Solicitation for fe80::4a4:3ff:fef9:32fa from f2:43:01:35:d5:89
Frame 1: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits) on interface 0
Linux cooked capture
Internet Protocol Version 6, Src: fc00:1::1, Dst: fc00:3::bb
    0110 .... = Version: 6
    .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 124
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 62
    Source: fc00:1::1
    Destination: fc00:3::bb
    Routing Header for IPv6 (Segment Routing)
        Next Header: IPIP (4)
        Length: 4
        [Length: 40 bytes]
        Type: Segment Routing (4)
        Segments Left: 1
        First segment: 1
        Flags: 0x00
        Reserved: 0000
        Address[0]: fc00:4::bb [next segment]
        Address[1]: fc00:3::bb
        [Segments in Traversal Order]
    Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.2.1
    Internet Control Message Protocol
```

图 7：R3 上抓包-END 操作

3.　R4

在 R4 上配置 SRv6。

```
"Node: r4"
root@ubuntu:~/topo# bash -x service_chain/
config/  topo.py
root@ubuntu:~/topo# bash -x service_chain/config/
B6_r1.sh  B6_r4.sh  dx4_r3.sh  dx6_r1.sh  sc_fw.sh  sc_r3.sh
B6_r3.sh  dx4_r1.sh  dx4_r4.sh  dx6_r4.sh  sc_r1.sh  sc_r4.sh
root@ubuntu:~/topo# bash -x service_chain/config/dx4_r
dx4_r1.sh  dx4_r3.sh  dx4_r4.sh
root@ubuntu:~/topo# bash -x service_chain/config/dx4_r4.sh
B6_r1.sh      B6_r4.sh      dx4_r3.sh     dx6_r3.sh     service_chain/
B6_r3.sh      dx4_r1.sh     dx6_r1.sh     dx6_r4.sh     simple.py
root@ubuntu:~/topo# bash -x service_chain/config/dx4_r4.sh
+ echo 'Adding DX4 Rules for R4'
Adding DX4 Rules for R4
+ ip -6 route add fc00:4::bb/128 encap seg6local action End.DX4 nh4 10.0.2.1 dev
 r4-eth1
+ ip route add 10.0.0.0/24 encap seg6 mode encap segs fc00:1::bb dev r4-eth1
root@ubuntu:~/topo#
```

图 8：R4 配置脚本

R4 上收到 R3 发来的数据包，由于 Segment Left 已经被 R3 更新为 0，R4 会根据策略执行 End.DX4 操作，去掉 IPv6 外层报头，转发到指定的 10.0.2.1 主机，从而完成了 VPNv4 以及流量工程。

```
No.  Time         Source                Destination           Protocol Length Info
   1 0.000000000  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3d71, seq=1/256, ttl=64 (no response found!)
   2 0.000009167  10.0.0.1              10.0.2.1              ICMP     100 Echo (ping) request  id=0x3d71, seq=1/256, ttl=63 (reply in 3)
   3 0.000018516  10.0.2.1              10.0.0.1              ICMP     100 Echo (ping) reply    id=0x3d71, seq=1/256, ttl=64 (request in 2)
   4 0.000021799  10.0.2.1              10.0.0.1              ICMP     164 Echo (ping) reply    id=0x3d71, seq=1/256, ttl=64
   5 1.026381736  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3d71, seq=2/512, ttl=64 (no response found!)
   6 1.026391294  10.0.0.1              10.0.2.1              ICMP     100 Echo (ping) request  id=0x3d71, seq=2/512, ttl=63 (reply in 7)
   7 1.026400415  10.0.2.1              10.0.0.1              ICMP     100 Echo (ping) reply    id=0x3d71, seq=2/512, ttl=64 (request in 6)
   8 1.026404306  10.0.2.1              10.0.0.1              ICMP     164 Echo (ping) reply    id=0x3d71, seq=2/512, ttl=64
   9 5.058115174  fe80::8879:efff:fef…  2001:14::1            ICMPv6    88 Neighbor Solicitation for 2001:14::1 from 8a:79:ef:fa:72:31
  10 5.058128036  fe:55:b6:6b:80:10                           ARP       44 Who has 10.0.2.1? Tell 10.0.2.2
  11 5.058124504  da:c5:4a:5a:e8:08                           ARP       44 Who has 10.0.2.2? Tell 10.0.2.1
  12 5.058254481  fe:55:b6:6b:80:10                           ARP       44 10.0.2.2 is at fe:55:b6:6b:80:10
  13 5.058135338  fe80::4a4:3ff:fef9:…  2001:34::2            ICMPv6    88 Neighbor Solicitation for 2001:34::2 from 06:a4:03:f9:32:fa
  14 5.058266255  2001:34::2            fe80::4a4:3ff:fef9:…  ICMPv6    80 Neighbor Advertisement 2001:34::2 (rtr, sol)

▸ Frame 1: 180 bytes on wire (1440 bits), 180 bytes captured (1440 bits) on interface 0
▸ Linux cooked capture
▾ Internet Protocol Version 6, Src: fc00:1::1, Dst: fc00:4::bb
    0110 .... = Version: 6
  ▸ .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
    .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
    Payload Length: 124
    Next Header: Routing Header for IPv6 (43)
    Hop Limit: 61
    Source: fc00:1::1
    Destination: fc00:4::bb
  ▾ Routing Header for IPv6 (Segment Routing)
      Next Header: IPIP (4)
      Length: 4
      [Length: 40 bytes]
      Type: Segment Routing (4)
      Segments Left: 0
      First segment: 1
    ▸ Flags: 0x00
      Reserved: 0000
      Address[0]: fc00:4::bb
      Address[1]: fc00:3::bb
    ▸ [Segments in Traversal Order]
  ▸ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.2.1
  ▸ Internet Control Message Protocol
```

图 8：R4 上抓包-END.DX4 操作

下图是执行过 End.DX4 操作后的数据包抓包情况，可见已经还原为原始的 IPv4 数据包。

```
No.  Time         Source                Destination           Protocol Length Info
   1 0.000000000  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3d71, seq=1/256, ttl=64 (no response found!)
   2 0.000009167  10.0.0.1              10.0.2.1              ICMP     100 Echo (ping) request  id=0x3d71, seq=1/256, ttl=63 (reply in 3)
   3 0.000018516  10.0.2.1              10.0.0.1              ICMP     100 Echo (ping) reply    id=0x3d71, seq=1/256, ttl=64 (request in 2)
   4 0.000021799  10.0.2.1              10.0.0.1              ICMP     164 Echo (ping) reply    id=0x3d71, seq=1/256, ttl=64
   5 1.026381736  10.0.0.1              10.0.2.1              ICMP     180 Echo (ping) request  id=0x3d71, seq=2/512, ttl=64 (no response found!)
   6 1.026391294  10.0.0.1              10.0.2.1              ICMP     100 Echo (ping) request  id=0x3d71, seq=2/512, ttl=63 (reply in 7)
   7 1.026400415  10.0.2.1              10.0.0.1              ICMP     100 Echo (ping) reply    id=0x3d71, seq=2/512, ttl=64 (request in 6)
   8 1.026404306  10.0.2.1              10.0.0.1              ICMP     164 Echo (ping) reply    id=0x3d71, seq=2/512, ttl=64
   9 5.058115174  fe80::8879:efff:fef…  2001:14::1            ICMPv6    88 Neighbor Solicitation for 2001:14::1 from 8a:79:ef:fa:72:31
  10 5.058128036  fe:55:b6:6b:80:10                           ARP       44 Who has 10.0.2.1? Tell 10.0.2.2
  11 5.058124504  da:c5:4a:5a:e8:08                           ARP       44 Who has 10.0.2.2? Tell 10.0.2.1
  12 5.058254481  fe:55:b6:6b:80:10                           ARP       44 10.0.2.2 is at fe:55:b6:6b:80:10
  13 5.058135338  fe80::4a4:3ff:fef9:…  2001:34::2            ICMPv6    88 Neighbor Solicitation for 2001:34::2 from 06:a4:03:f9:32:fa
  14 5.058266255  2001:34::2            fe80::4a4:3ff:fef9:…  ICMPv6    80 Neighbor Advertisement 2001:34::2 (rtr, sol)

▸ Frame 2: 100 bytes on wire (800 bits), 100 bytes captured (800 bits) on interface 0
▸ Linux cooked capture
▾ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.2.1
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  ▸ Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 84
    Identification: 0xe3c2 (58306)
  ▸ Flags: 0x4000, Don't fragment
    Time to live: 63
    Protocol: ICMP (1)
    Header checksum: 0x41e5 [validation disabled]
    [Header checksum status: Unverified]
    Source: 10.0.0.1
    Destination: 10.0.2.1
▸ Internet Control Message Protocol
```

图 9：R4 上抓包-END.DX4 操作后的数据包

下图是 Ping 回程数据包抓包情况，可见加了 IPv6 报头，目的地址是 fc00:1:bb，但 SRH 中没有 Segment，这是因为 Segment Left=0（此时其实不需要 SRH，具体见参考文献 2）。

图 10：R4 上抓包-Ping 回程数据包

互通验证结果：

```
"Node: a"
root@ubuntu:~/topo# ping 10.0.2.1
PING 10.0.2.1 (10.0.2.1) 56(84) bytes of data.
64 bytes from 10.0.2.1: icmp_seq=1 ttl=63 time=0.076 ms
64 bytes from 10.0.2.1: icmp_seq=2 ttl=63 time=0.094 ms
64 bytes from 10.0.2.1: icmp_seq=3 ttl=63 time=0.098 ms
64 bytes from 10.0.2.1: icmp_seq=4 ttl=63 time=0.103 ms
64 bytes from 10.0.2.1: icmp_seq=5 ttl=63 time=0.086 ms
64 bytes from 10.0.2.1: icmp_seq=6 ttl=63 time=0.087 ms
64 bytes from 10.0.2.1: icmp_seq=7 ttl=63 time=0.243 ms
64 bytes from 10.0.2.1: icmp_seq=8 ttl=63 time=0.098 ms
64 bytes from 10.0.2.1: icmp_seq=9 ttl=63 time=0.087 ms
64 bytes from 10.0.2.1: icmp_seq=10 ttl=63 time=0.085 ms
```

图 11：主机 a Ping 主机 b 结果

5.2 Linux SRv6 实现服务链+流量工程

5.2.1 概述

目的：使 2 台仅支持 IPv4 的主机（主机 a 和主机 b），通过 SRv6 实现互通，并实现服务链+流量工程。本文中我们首先验证支持 SRv6 的服务（SR-aware），不支持 SRv6 的服务（Non SR-aware）将在第二篇中介绍。

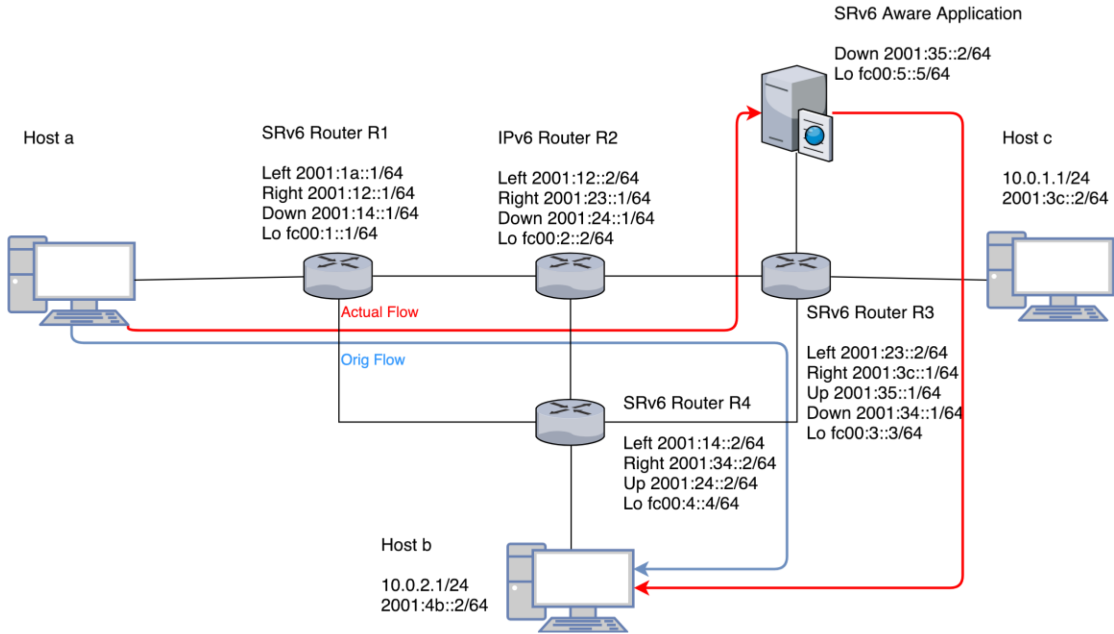

图 12：Linux SRv6 实现服务链+流量工程拓扑

拓扑和第一个场景类似，路由器 R1、R3 和 R4 为支持 SRv6 的路由器，R2 为仅支持 IPv6 的路由器，通过开源 Quagga 实现静态路由，路由器与路由器之间仅通过 IPv6 互通。R3 路由器上面加入了一台支持 SRv6 的 IDS 设备（Snort）。

路由器的 Loopback 地址间通过 IPv6 路由可达。

主机 a 到主机 b 的流量，在 R1 上新增的 SRH 要求经由 R3 进行转发，因此原流量路径如图蓝色路径所示。

在 R3 路由器上，修改 End 操作为 End.B6.Encaps 操作，将流量先引导到 IDS 进行处理，再回到 R3 进行正常转发流程。对应的新的流量路径如图红色路径所示。

End.B6.Encaps 的功能我们之前已经简单的进行了介绍，该操作在现有 IPv6 数据包上封装一个新的 IPv6 的报头，并添加新的 SRH 报头；而 End.B6 操作则不会添加新的 IPv6 报头，而是直接插入新的 SRH 报头到现有的 IPv6 报头当中。这两种操作本质上都相当于是 Binding-SID。

具体数据包转发流程中报头的变化可参见图 13 和图 14，图中报文上的数字表示数据包在网络中的转发顺序。

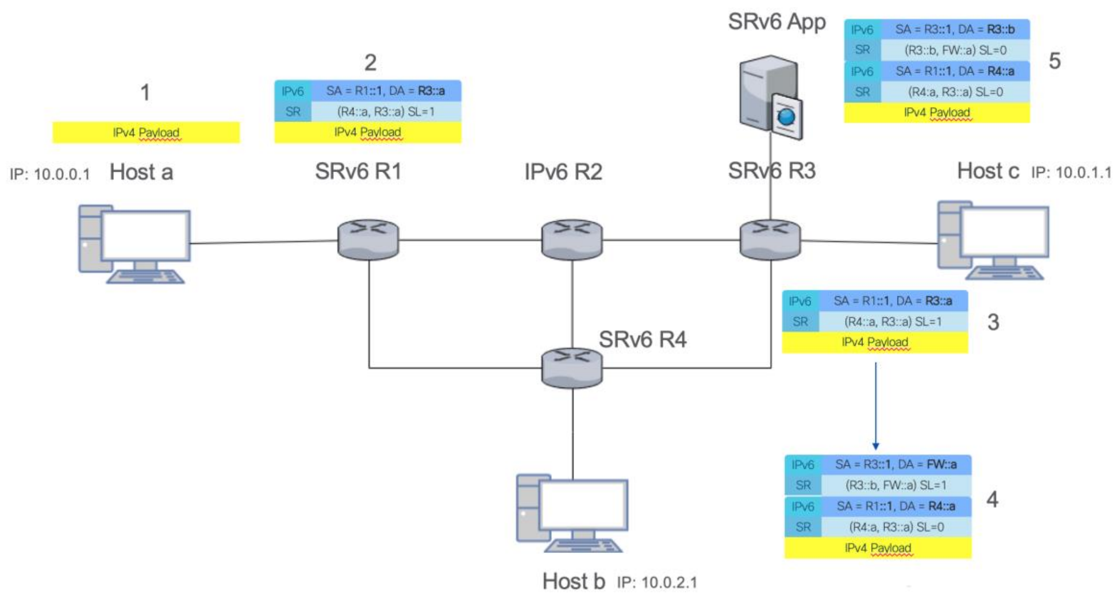

图 13：Linux SRv6 实现服务链+流量工程-数据包转发流量(Part1)

图 13 为数据包的前半部分转发流程，主机 a 发出 IPv4 数据包，R1 对 IPv4 数据包进行封装，加入 SRv6 报头，其中包含 2 个 Segment。数据包首先转发到 R3，执行 R3::a 对应的操作。我们在 R3 定义 R3::a 操作为 End.B6.Encaps，变化如图右下角，首先 Segment Left 减 1，更新 IPv6 目的地址，然后在数据包外层增加新的 IPv6 报头，SRH 中添加 2 个 Segment，首先转发到 IDS，然后返回 R3。

当数据包到达 IDS 之后，由于 IDS 支持 SRv6，会执行 End 操作，并检查/过滤数据包内容，当检查完成后进行正常转发，End 操作更新最外层 IPv6 报头，Segment Left 减 1，目的地址修改为 R3::b。

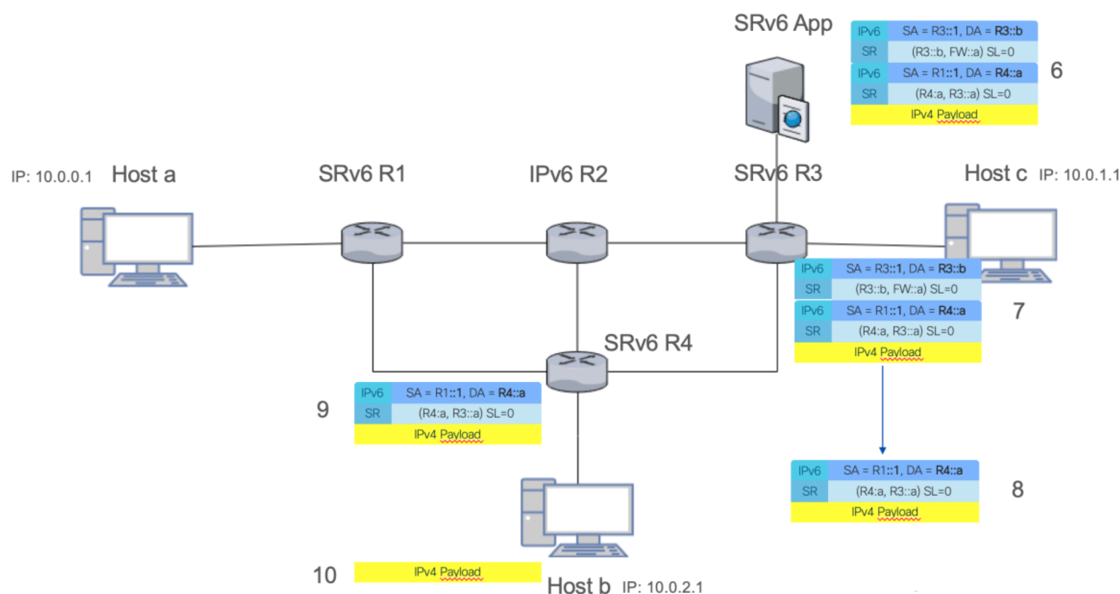

图 14：Linux SRv6 实现服务链+流量工程-数据包转发流量(Part2)

图 14 表示接下来的转发流程。当数据包第二次返回 R3 时，由于 Segment 为 R3::b，R3 将执行不同的操作，在这个场景中为 End.DX6。End.DX6 操作会将外层的 IPv6 报头去掉，然后正常转发，变化如图中所示。

R4 收到数据包之后，会根据 Segment 执行 End.DX4 操作，去掉 IPv6 报头，将 IPv4 数据包转发给主机 b，完成整个转发流程。

从上述过程可以看出，SR 实现服务链本质上是基于自身的流量工程能力，事实上服务节点（或服务代理）本质上只是 Segment 列表中的一个 Segment，这不单适用于本例中支持 SR 的服务，也适用于不支持 SR 的服务。

5.2.2 具体步骤

1. 下面我们启动拓扑

```
sudo python topo2.py
```

Python 脚本会自动配置好设备初始配置以及地址。

2. 配置 R1

配置 T.Encaps 操作，为 IPv4 数据包添加 IPv6 报头和 Segment

```
ip route add 10.0.2.0/24 encap seg6 mode encap  segs fc00:3::a,fc00:4::
a dev r1-eth1
```

为回程数据包执行 End.DX4 操作，发给主机 a

```
ip -6 route add fc00:1::a/128 encap seg6local action End.DX4 nh4 10.0.0.
1 dev r1-eth1
```

3. 配置 R3

执行 End.B6.Encaps 操作，添加新的 SRH 去往 IDS 实现服务链

```
ip -6 route add fc00:3::a/128 encap seg6local action End.B6.Encaps srh segs fc00:5::a,fc00:3::b dev r3-eth1
```

对执行完服务链操作的数据包，去掉外层 IPv6 报头，进行常规的 IPv6 转发

```
ip -6 route add fc00:3::b/128 encap seg6local action End.DX6 nh6 :: dev r3-eth1
```

4. 配置 IDS

执行 End 操作，IDS 的内部应用对数据包进行检测

```
ip -6 route add fc00:5::a/128 encap seg6local action End dev ids-eth0
```

5. 最后配置 R4

对收到的数据包执行 End.DX4 操作，去掉 IPv6 报头，正常转发给主机 b

```
ip -6 route add fc00:4::a/128 encap seg6local action End.DX4 nh4 10.0.2.1 dev r4-eth1
```

回程路由直接去往 R1

```
ip route add 10.0.0.0/24 encap seg6 mode encap  segs fc00:1::a dev r4-eth1
```

配置完成后，从主机 a 可以 Ping 通主机 b，并且在 IDS 处可以监测到主机 a 发往主机 b 的数据包。

5.2.3 脚本执行及抓包验证

为了方便，我们将相关的配置都通过 Python 脚本实现了自动化。代码见 https://github.com/ljm625/srv6_Sandbox

1. R1

在 R1 上配置 SRv6。

图 15：R1 配置脚本

2. R3

在 R3 上配置 SRv6。

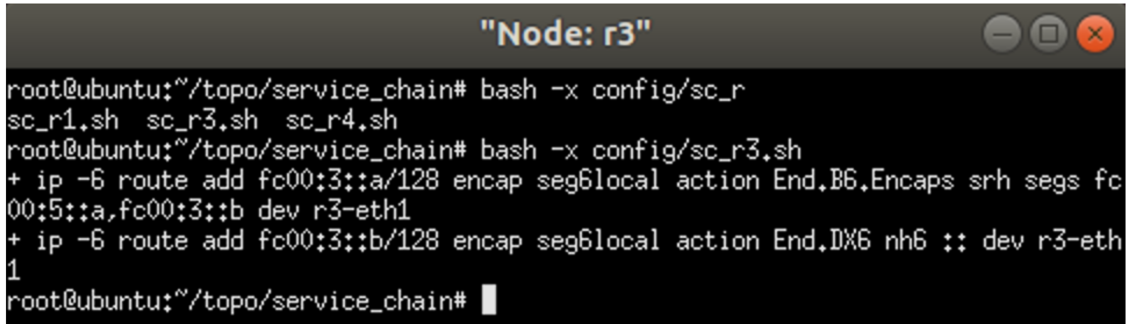

图 16：R3 配置脚本

3. IDS

在 IDS 上配置 SRv6。

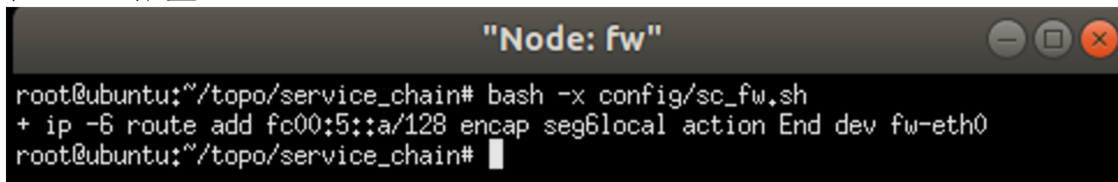

图 17：IDS 配置脚本

4. R4

在 R4 上配置 SRv6。

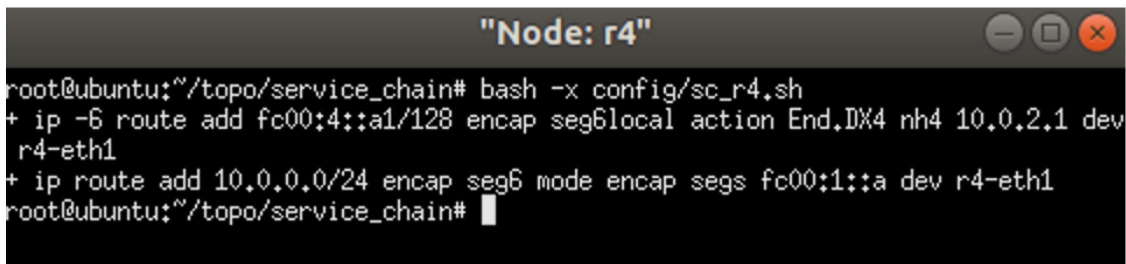

图 18：R4 配置脚本

我们从主机 a(10.0.0.1) Ping 主机 b(10.0.2.1)进行测试。

在主机 a 上 Ping 主机 b 的情况：

```
"Node: a"
root@ubuntu:~/topo# ping 10.0.2.1
PING 10.0.2.1 (10.0.2.1) 56(84) bytes of data.
64 bytes from 10.0.2.1: icmp_seq=1 ttl=63 time=0.170 ms
64 bytes from 10.0.2.1: icmp_seq=2 ttl=63 time=0.096 ms
64 bytes from 10.0.2.1: icmp_seq=3 ttl=63 time=0.076 ms
64 bytes from 10.0.2.1: icmp_seq=4 ttl=63 time=0.080 ms
64 bytes from 10.0.2.1: icmp_seq=5 ttl=63 time=0.082 ms
64 bytes from 10.0.2.1: icmp_seq=6 ttl=63 time=0.081 ms
^C
--- 10.0.2.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5100ms
rtt min/avg/max/mdev = 0.076/0.097/0.170/0.034 ms
root@ubuntu:~/topo#
```

图 19：主机 a 可以 ping 通主机 b

在 R3 上的抓包情况：

```
477 7028.4337161… 10.0.0.1    10.0.2.1    ICMP    258 Echo (ping) request  id=0x354a, seq=1/256, ttl=64 (n
478 7028.4337273… 10.0.0.1    10.0.2.1    ICMP    258 Echo (ping) request  id=0x354a, seq=1/256, ttl=64 (n
479 7029.4421359… 10.0.0.1    10.0.2.1    ICMP    258 Echo (ping) request  id=0x354a, seq=2/512, ttl=64 (n
480 7029.4421498… 10.0.0.1    10.0.2.1    ICMP    258 Echo (ping) request  id=0x354a, seq=2/512, ttl=64 (n
481 7030.4658506… 10.0.0.1    10.0.2.1    ICMP    258 Echo (ping) request  id=0x354a, seq=3/768, ttl=64 (n

Frame 477: 258 bytes on wire (2064 bits), 258 bytes captured (2064 bits) on interface 0
Ethernet II, Src: 36:17:43:42:e2:2d (36:17:43:42:e2:2d), Dst: c6:46:5d:50:19:39 (c6:46:5d:50:19:39)
Internet Protocol Version 6, Src: fc00:3::3, Dst: fc00:5::a
  0110 .... = Version: 6
  .... 0000 0000 .... .... .... .... .... = Traffic Class: 0x00 (DSCP: CS0, ECN: Not-ECT)
  .... .... .... 0000 0000 0000 0000 0000 = Flow Label: 0x00000
  Payload Length: 204
  Next Header: Routing Header for IPv6 (43)
  Hop Limit: 61
  Source: fc00:3::3
  Destination: fc00:5::a
  Routing Header for IPv6 (Segment Routing)
    Next Header: IPv6 (41)
    Length: 4
    [Length: 40 bytes]
    Type: Segment Routing (4)
    Segments Left: 1
    First segment: 1
    Flags: 0x00
    Reserved: 0000
    Address[0]: fc00:3::b [next segment]
```

图 20：R3 上抓包-执行完 End.B6 操作之后的数据包

在 IDS 上的抓包情况：

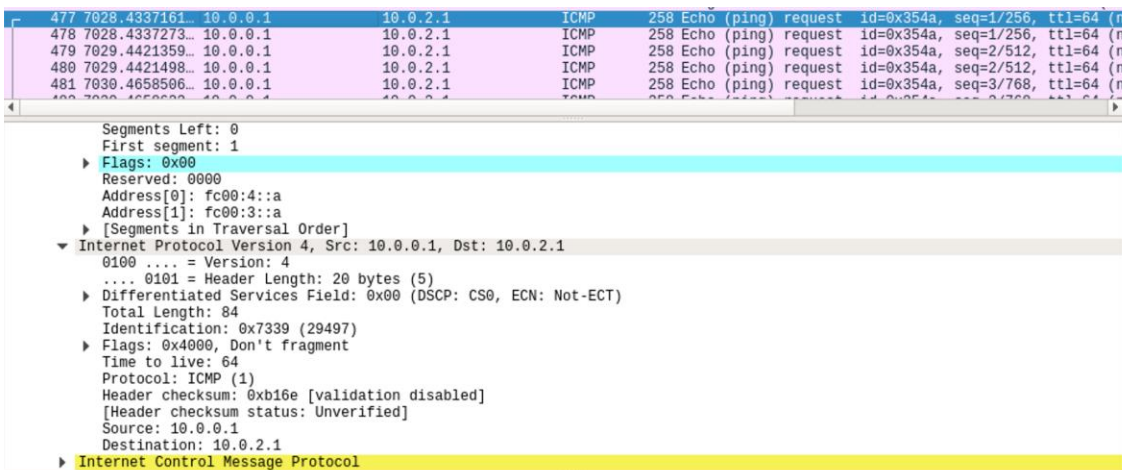

图 21：IDS 上抓包-IDS 应用执行完操作,准备发给 R3 的数据包

我们通过在 IDS 上的 Snort 进行监测：

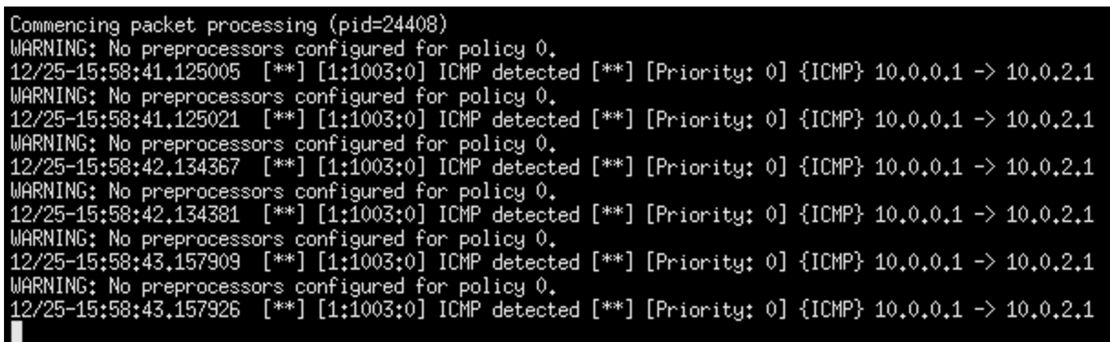

图 22：Snort 监测结果

Snort 规则配置为对从 10.0.0.1 去往 10.0.2.1 的 ICMP 包进行告警，可以看到 Snort 可以正确检测到从 10.0.0.1 去往 10.0.2.1 的 ICMP 包。

六、总结与展望

本文基于两个常见的应用场景（VPN+流量工程，服务链+流量工程）测试了 Linux 内核对 SRv6 的支持情况，测试了常用的 End 操作，包括 End、End.DX4、End.B6.Encaps 和 T.Encaps。

从测试结果来看，Linux 内核已经能很好地支持 SRv6 常用操作，测试结果令人满意。由于条件所限，本文并未进行性能测试。

在下一篇文章中，我们将测试 SRv6 End.AD 等操作-即使服务不支持 SRv6，仍然可以通过 End.AD 操作实现服务链。

为了提高性能，业界越来越多使用 FD.IO，因此后续文章中我们也会介绍采用 FD.IO 来实现 SRv6，并与网络设备结合，构建高性能的虚拟/物理一体、Overlay/Underlay 融合的高性能 SRv6 网络。

【参考文献】

1. SRH draft: https://tools.ietf.org/html/draft-ietf-6man-segment-routing-header-15

2. SRv6 draft：https://tools.ietf.org/html/draft-filsfils-spring-srv6-network-programming-06

3. Segment Routing 的相关资料：https://segment-routing.net

4. SRv6 Linux 的相关资料/教程：https://segment-routing.org

5. SRv6 VPP 的实现和教程：
https://wiki.fd.io/view/VPP/Segment_Routing_for_IPv6