使用tun虚拟网络接口建立IP隧道的实例

---- Written by whowin

> 通常的socket编程，面对的都是物理网卡，Linux下其实很容易创建虚拟网卡；本文简单介绍一下Linux虚拟网卡的概念，并以tun设备为例在客户端和服务器端分别建立一个实际的虚拟网卡，最终实现一个从客户端到服务器的简单的IP隧道，希望本文能对理解虚拟网卡和IP隧道有所帮助，本文将提供完整的源程序；阅读本文需要具备在Linux下使用C语言进行IPv4下socket编程的基本能力，本文对网络编程的初学者难度较大。

1. Linux下的虚拟网卡TUN/TAP

- TUN和TAP是Linuxn内核的虚拟网络设备，不同于普通靠硬件网络适配器实现的设备，这些虚拟的网络设备全部用软件实现，并可以向运行于Linux上的应用软件提供与硬件的网络设备完全相同的功能；
- TAP等同于一个以太网设备，它操作OSI模型的第二层(数据链路层)数据包，通常我们所使用的网络就是以太网数据帧，所以要使用TAP设备，就需要自己构建以太网报头、IP报头、TCP/UDP报头；
- TUN模拟了网络层设备，操作第三层(网络层)数据包，通常我们使用的TCP/UDP报文在网络层使用的IP协议，所以使用TUN设备，需要自己构建IP报头和TCP/UDP报头，比TAP设备少构建一个以太网报头；
- Linux通过TUN/TAP设备向绑定该设备的用户空间的应用程序发送数据；同样，用户空间的应用程序也可以像操作硬件网络设备那样，通过TUN/TAP设备发送数据；在后面这种情况下，TUN/TAP设备向Linux的网络协议栈提交数据包，从而模拟从外部接收数据的过程；

2. 构建一个TUN设备

- 上一节的描述显然过于枯燥，可能会对初次接触虚拟网卡的读者感到困惑，不知所云，本节将实际建立一个tun设备，帮助你走出困惑；
- 构建一个基本的tun设备，只需要两个步骤
  1. **编写一个程序，至少完成三个任务**
     - 以可读写模式打开设备文件 `/dev/net/tun`

```
int fd;
fd = open("/dev/net/tun", O_RDWR));
```

     - 向Linux内核注册一个tun设备名称，本例中为tun0

> (struct ifreq)定义在头文件<linux/if.h>中，在我的很多文章中都有介绍，比如文章《如何使用raw socket发送UDP报文》，如果需要，可以参考；

```
struct ifreq ifr;
memset(&ifr, 0, sizeof(ifr));
ifr.ifr_flags = IFF_TUN | IFF_NO_PI;
strcpy(ifr.ifr_name, "tun0");

ioctl(fd, TUNSETIFF, (void *)&ifr);
```

     - 编写处理tun0接收/发送数据的程序

```
char buffer[BUFSIZE];

while (1) {
  read(fd, buffer, BUFSIZE);
  // todo
}
```

2. **为设备分配IP地址**(本例中为tun0分配的IP为10.0.0.1)

```
sudo ifconfig tun0 10.0.0.1 netmask 255.255.255.0 up
```

- 把上面的代码片段组合在一起，就可以完成一个tun设备的建立，文件名：tun_01.c(**点击文件名下载源文件**)
- 这段程序在进入循环前增加了 `system("ifconfig tun0 10.0.0.1/24 up")`，为tun0分配了IP地址10.0.0.1，所以运行完后就不需要再为这个设备分配IP了；
- 编译：`gcc -Wall tun-01.c -o tun-01`
- 该程序需要root权限运行，主要是因为其中使用了ioctl，运行：`sudo ./tun-01`
- 运行该程序，会构建一个tun设备，打开一个新的终端，使用 `ifconfig` 将可以看到系统中多了一个虚拟网络接口tun0，使用 `route -n` 查看路由也会看到增加了一条关于tun0设备的路由

```
whowin@whowin-ubuntu:~$ sudo ifconfig tun0 10.0.0.1 netmask 255.255.255.0 up
whowin@whowin-ubuntu:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.2.114  netmask 255.255.255.0  broadcast 192.168.2.255
        inet6 fe80::a00:27ff:fef1:b92  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:f1:0b:92  txqueuelen 1000  (以太网)
        RX packets 91264  bytes 22157879 (22.1 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 77731  bytes 26294141 (26.2 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (本地环回)
        RX packets 198714  bytes 50148506 (50.1 MB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 198714  bytes 50148506 (50.1 MB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

tun0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1500
        inet 10.0.0.1  netmask 255.255.255.0  destination 10.0.0.1
        inet6 fe80::164d:452a:d307:112  prefixlen 64  scopeid 0x20<link>
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (未指定)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 2  bytes 96 (96.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

whowin@whowin-ubuntu:~$ route -n
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         192.168.2.3     0.0.0.0         UG    100    0        0 enp0s3
10.0.0.0        0.0.0.0         255.255.255.0   U     0      0        0 tun0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0        0 enp0s3
192.168.2.0     0.0.0.0         255.255.255.0   U     100    0        0 enp0s3
whowin@whowin-ubuntu:~$
```

图1：构建一个tun设备后

- 尽管建立起了虚拟网卡tun0，但因为程序过于简单，所以这样建立的设备什么事情都做不了，必须完善程序，才能让这个设备真正地发挥作用；

- tun设备是一个第三层(网络层)的设备，在这个设备上只能收到IP报头，收不到以太网报头，所以Linux索性没有为tun设备分配MAC地址；
- 后面将以本节的程序为基础，不断改进，最终写出一个简单的IP隧道的程序。

3. 使用tun设备的基本数据流向

- 设备建立起来以后，程序员关心的是我们如何从这个设备上收发报文，如何处理这些报文；
- 对于一个物理网络接口而言，接口一端连接着网络协议栈，另一端连接着物理网络；而对于一个虚拟网络接口而言，接口的一端仍然连接着网络协议栈，但是另一端连接着一个应用程序，也就是我们前面下载的那个程序(tun-01.c)，我们把这个程序称为 **application-tun**；
- 可以和一个物理网络接口比较来说明虚拟网络接口的数据流向，在物理接口上要发送到物理网络上去的报文，相对于虚拟接口将被发送到应用程序 **application-tun** 上；
- 当我们使用socket发送报文时，报文被提交给Linux的网络协议栈，协议栈为报文封装各个协议层的头，并根据路由表将报文交给相应设备的驱动程序，比如enp0s3的驱动程序，然后由驱动程序将报文发送到物理网络上(物理设备)，或者发送给应用程序 application-tun(虚拟设备)；
- 在上一节中，我们使用 `route -n` 已经看到了关于tun0设备的路由：

```
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         192.168.2.3     0.0.0.0         UG    100    0      0  enp0s3
10.0.0.0        0.0.0.0         255.255.255.0   U     0      0      0  tun0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0      0  enp0s3
192.168.2.0     0.0.0.0         255.255.255.0   U     100    0      0  enp0s3
```

- 路由表明，当目的IP地址为10.0.0.x时，报文将被送到虚拟设备tun0的驱动程序上去，该设备绑定的IP为10.0.0.1；
- 还有一条路由，当目的IP地址为192.168.2.x时，报文将被送到物理设备enp0s3的驱动程序上去，该设备绑定的IP为192.168.2.114；
- 这两条路由比较相似，区别是一个是物理设备enp0s3，另一个是虚拟设备tun0，我们拿这两条路由进行对比说明数据流向；
- **发送报文到物理/虚拟接口绑定的IP地址上**
  - 当我们发送一个UDP报文到 **192.168.2.114:5678**(也就是本机物理设备enp0s3的IP)时，根据路由，报文被送给enp0s3的驱动程序，驱动程序并不会把这个报文发送到物理网络上，因为enp0s3的驱动程序已经是这个报文最终的目的地，所以enp0s3的驱动程序会将这个报文发到一个正在监听192.168.2.114:5678的用户程序上，如果我们没有编写这个程序，报文将被丢弃，这样我们就收不到这个报文；
  - 当我们发送一个UDP报文到 **10.0.0.1:5678**(也就是本机虚拟设备tun0的IP)时，根据路由，报文被送给tun0的驱动程序，驱动程序并不会把这个报文发送到 application-tun 上，因为tun0的驱动程序已经是这个报文的最终目的地，所以tun0的驱动程序会将这个报文发到一个正在监听10.0.0.1:5678的用户程序上，和物理设备一样，如果我们没有编写这个程序，报文将被丢弃，我们收不到这个报文；

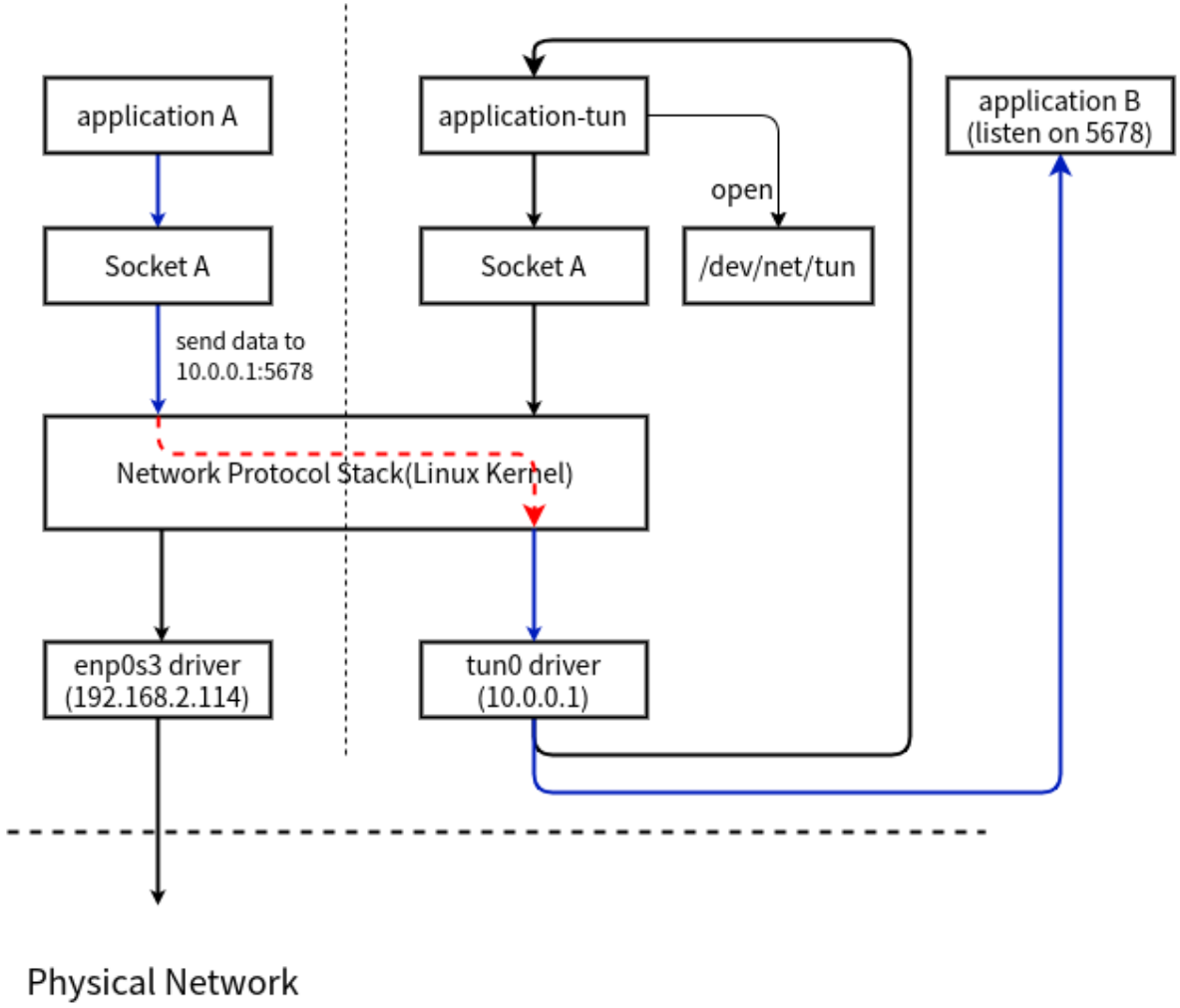

图2：发送报文到tun0的IP上

- **发送报文到符合路由的其他IP地址上**
  - 当我们发送一个UDP报文到 **192.168.2.112:5678** 时，根据路由报文会被送给enp0s3的驱动程序，驱动程序会把这个报文发送到物理网络上；
  - 当我们发送一个UDP报文到 **10.0.0.2:5678** 时，根据路由报文会被送给报文被送给tun0的驱动程序，驱动程序会把这个报文发送到应用程序 **application-tun** 上；

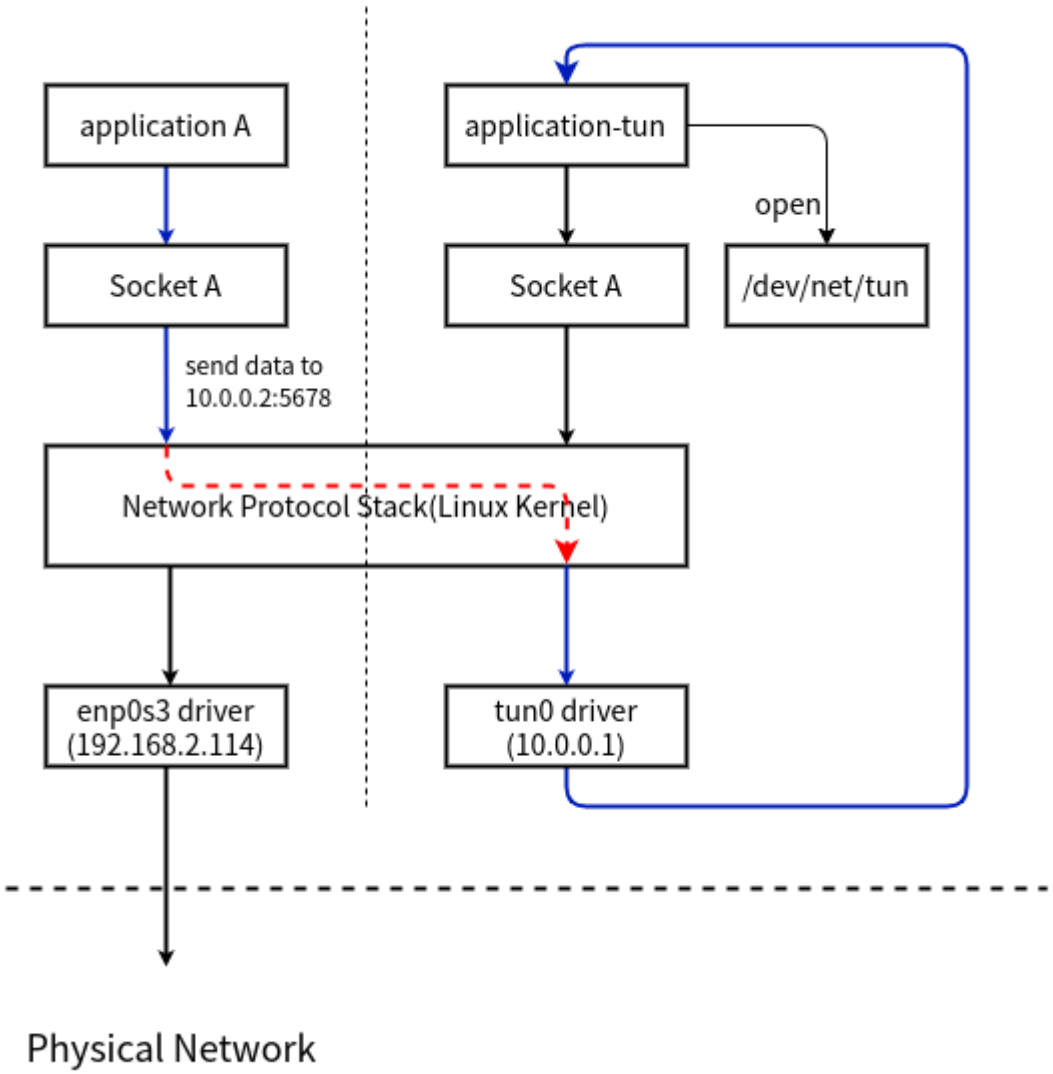

图3：发送报文到符合tun0路由的其他IP上

- **对上述说明可以做一个简单的测试**
  - 打开终端，运行前面的程序：tun-01

    ```
    sudo ./tun-01
    ```

  - 打开另一个终端，使用下面命令分别向 **10.0.0.1:5678** 发送数据，在运行 tun-01 的终端上并不会显示收到数据；

    ```
    echo "hello" > /dev/udp/10.0.0.1/5678
    ```

  - 使用下面命令分别向 **10.0.0.2:5678** 发送数据，在运行 tun-01 的终端上会显示收到数据；

    ```
    echo "hello" > /dev/udp/10.0.0.2/5678
    ```

- **源IP地址的选择**
  - 当我们在电脑系统上运行 `sudo ./tun-01` 时，我们的系统就有了两个IP地址，一个是物理网卡的，IP为192.168.2.114，另一个是虚拟网卡的，IP为10.0.0.1；

- 当我们在做上面的测试时，我们用 `echo ......` 命令向 10.0.0.1 和 10.0.0.2 发送了UDP消息，发送时我们并没有指定源IP地址，那么发出的消息的源IP地址是什么呢？192.168.2.114 还是 10.0.0.1？
- 我们把前面那个程序 tun_01.c 改一下，一是增加一些错误判断，使这个程序更加完善一些，另外我们增加一个显示IPv4报头的功能，这样我们就可以看到IP头中的源IP地址了；
- 改好的程序文件名为：tun-02.c(**点击文件名下载源程序**)
- 编译：`gcc -Wall tun-02.c -o tun-02`
- 下面我们做个测试，向 10.0.0.2:5678 发送一条UDP消息，我们看看源IP地址是什么？
  - 打开一个终端，运行tun-02

    ```
    sudo ./tun-02
    ```

  - 打开另一个终端，向10.0.0.2发送消息

    ```
    echo "hello" > /dev/udp/10.0.0.2/5678
    ```

  - 在运行tun-02的终端上显示出源IP地址为10.0.0.1

    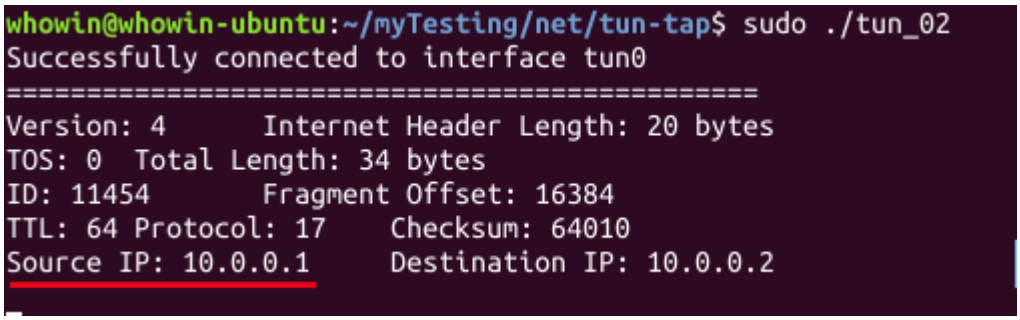

图4：Linux在多网卡环境下选择源IP

- 当使用sendmsg()发送数据时，是可以显式地指定源IP地址的；
- 路由表中有一个src字段，当没有指定源IP地址时，将使用选定路由的src字段作为源IP地址，使用 `ip route` 可以看到src字段

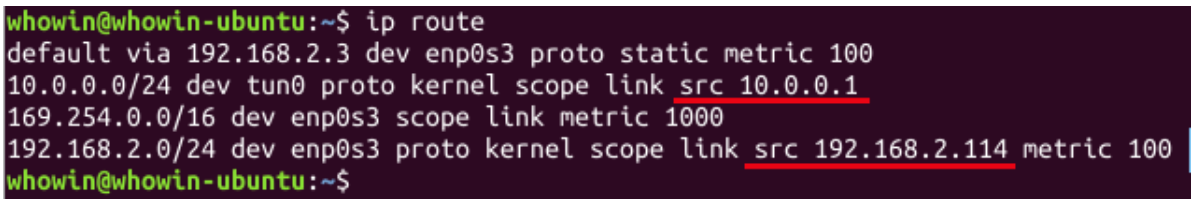

图5：ip route命令显示路由表中src字段

- 如果选定的路由没有src字段，Linux会搜寻选定路由的网络接口上所有绑定的IP，对IPv6将选择第一个搜寻到的地址，对IPv4则尽量选择与目标IP在同一网段的IP地址；

4. 使用tun设备搭建一个简单的IP隧道

- tun实际上是tunnel的前面三个字母，tun设备注定和隧道是有关系的，tun设备也的确常用来构建一个IP隧道；
- IP报文其实是指：IP报头 + TCP/UDP报头 + 数据

- 所谓IP隧道是指把一个IP报文作为数据再封装一个TCP头和IP头，所以整个报文变成：IP报头 + TCP报头 + (IP报头 + TCP/UDP报头 + 数据)
- 至于IP隧道的意义、应用场景之类的，本文不予讨论，可以自己去百个度或者谷个歌查一下，本文将致力于做一个简单的IP隧道；
- 先看一张示意图

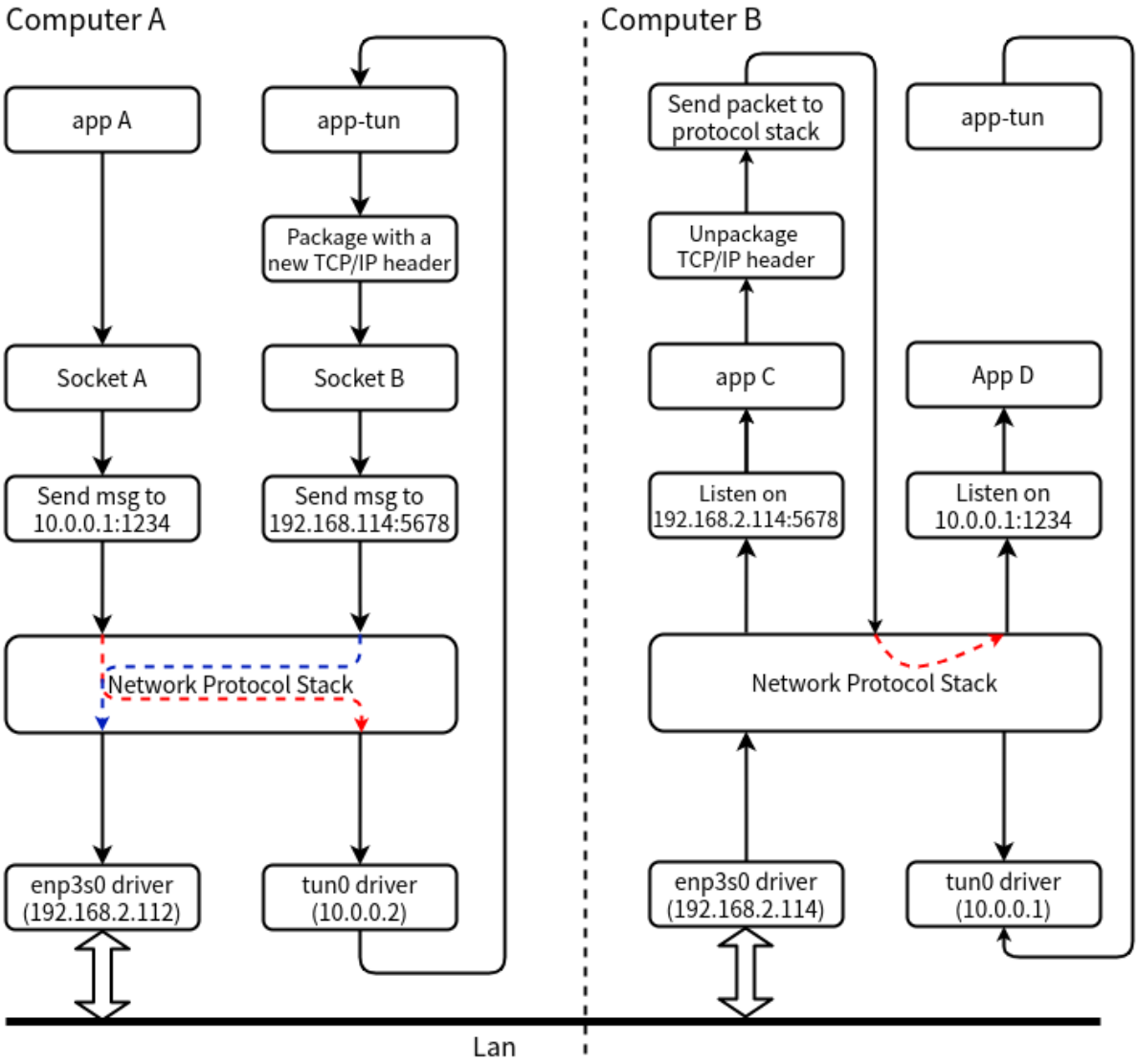

图6：简单的IP隧道示意图

- 有两台电脑，Computer A和Computer B
  - Computer A：
    1. 物理网卡为enp0s3，绑定IP：192.168.2.112
    2. 虚拟网卡为tun0，绑定IP：10.0.0.2
  - Computer B：
    1. 物理网卡为enp0s3，绑定IP：192.168.2.114
    2. 虚拟网卡为tun0，绑定IP：10.0.0.1
- Computer A和Computer B的路由表一样，如下：

```
内核 IP 路由表
目标            网关            子网掩码        标志  跃点   引用  使用 接口
0.0.0.0         192.168.2.3     0.0.0.0         UG    100    0      0  enp0s3
10.0.0.0        0.0.0.0         255.255.255.0   U     0      0      0  tun0
169.254.0.0     0.0.0.0         255.255.0.0     U     1000   0      0  enp0s3
192.168.2.0     0.0.0.0         255.255.255.0   U     100    0      0  enp0s3
```

- Computer A的应用程序app A向10.0.0.1:1234发送报文，Computer B的应用程序app D侦听在10.0.0.1:1234上；
- 目标很简单，computer A的app A直接向10.0.0.1:1234发送报文，computer B的app D能够正常收到收到，就像在一个局域网上一样；
- 首先要明确的，物理局域网的网段是192.168.2.x，所以向10.0.0.1发送报文并不会被送到物理局域网上，按照路由，这条报文会被送到tun0的驱动程序上去，因为10.0.0.1并不是computer A的虚拟网卡tun0绑定的IP，所以驱动程序会把这个报文送到application-tun上，所以如果我们不做处理，这个报文根本无法到达目的地；
- 如何处理这个报文使其发送到computer B的app D上去呢？通常的方法就是在computer A和computer B的物理网卡之间建立一条IP隧道；
- 当computer A启动applition-tun时，主动发起向computer B的连接，端口号定为5678，computer B在启动applition-tun时，主动侦听在端口5678上，并等待computer A的连接请求，一旦连接建立，这个隧道就建好了；
- computer A的application-tun收到发往10.0.0.1的报文时，要在整个IP报文上再包装上一个IP报头+TCP报头，TCP报头中指定目的端口号为5678，IP报头中指定目的IP为192.168.2.114，源IP为192.168.2.112，然后把这个新报文从建立的隧道中发出；
- computer B上侦听在5678端口上的应用程序app C会收到这个报文，app C去掉IP报头和TCP报头，把数据部分作为一个完整的报文重新从socket发出，这个报文的内容正是computer A发出的原始报文，computer B的内核协议栈根据路由会将该报文发给tun0的驱动程序，驱动程序会将这个报文送到正在侦听1234端口的app D上；
- 在客户端(computer A)需要编写一个程序，程序文件名：app-client.c，这个程序应遵循以下处理流程：
    1. 打开 /dev/net/tun 文件，返回tun_fd，在内核注册虚拟设备 tun0；
    2. 创建socket，sock_fd，在这个 sock_fd 上连接服务器端(computer B)的5678端口，建立IP隧道；
    3. 使用select检查tun_fd和sock_fd，并分别处理在这两个 fd 上收到的数据；
    4. 在tun_fd上收到数据的处理流程
        - 将收到的包括IP报头在内的报文作为数据从sock_fd上发出
    5. 在sock_fd上收到数据的处理流程
        - 把收到的数据作为一个IP报文显示报头及内容
- 在服务器端(computer B)编写一个程序，文件名为：app-server.c，这个程序应遵循以下处理流程：
    1. 打开 /dev/net/tun 文件，返回tun_fd，在内核注册虚拟设备 tun0；
    2. 创建socket，fd为sock_fd，在这个 sock_fd 上侦听5678端口，等待客户端连接以建立IP隧道；
    3. 接受客户端的连接请求，为新连接创建socket，fd为net_fd
    4. 使用select检查tun_fd和net_fd，并分别处理在这两个 fd 上收到的数据；
    5. 在tun_fd上收到数据的处理流程
        - 将收到的包括IP报头在内的报文作为数据从net_fd上发出
    6. 在net_fd上收到数据的处理流程

        - 把收到的数据(不包括IP报头和TCP报头)作为带有IP报头的报文发到tun_fd上
- 客户端程序：app-client.c(**点击文件名下载源程序**)
- 客户端程序编译：`gcc -Wall app-client.c -o app-client`
- 服务器端程序：app-server.c(**点击文件名下载源程序**)
- 服务器端程序编译：`gcc -Wall app-server.c -o app-server`
- 为了运行方便，也可以将这两个程序写成守护进程，将程序中注释掉的 `daemon(0, 0)` 放开即可；
- 请根据实际情况调整程序中的宏定义，SERVER_IP和TUN_IP；
- 在服务器端注意防火墙设置，打开5678端口或者关闭防火墙；
- 这两个程序的运行均需要root权限。
- 客户端程序测试
    - 需要打开三个终端窗口；
    - 首先将程序中的SERVER_IP改为本机的IP地址，然后重新编译；
    - 打开第一个终端，运行 `nc -l 5678`，这个命令将监听本机的5678端口；
    - 打开第二个终端，运行客户端程序：`sudo ./app-client`，应该显示"Connected to server ..."
    - 打开第三个终端，运行 `echo "hello" > /dev/udp/10.0.0.1/1234`，这个命令将向10.0.0.1的1234端口发送一个UDP报文，报文的数据部分为"hello"
    - 此时在第二个终端上应该显示"Received data from tun"，在第一个终端上收到一些乱码，但其中有"hello"字符串，乱码是因为我们收到的数据包括IP报头和UDP报头，这两部分是二进制的数据；
    - 如果你看到的和上面的描述一致，那么你的客户端程序基本没有问题；
    - 下面是截屏

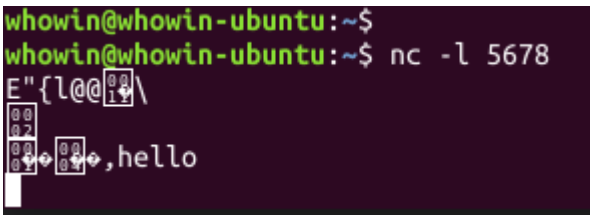

图7：测试客户端程序时的第一个终端

```
![screenshot of 2nd terminal for client test][img08]

<center><b>图8：测试客户端程序时的第二个终端</b></center>
```

- 服务器端程序无需独立测试；
- IP隧道测试
    - 需要两台机器，一台做客户端，另一台做服务器端
    - 再次强调，请根据实际情况调整程序中的宏定义，SERVER_IP和TUN_IP，并重新编译程序；
    - 在服务器端注意防火墙设置，打开5678端口或者关闭防火墙；
    - 在服务器端和客户端均需要打开两个终端，下面是测试方法示意图

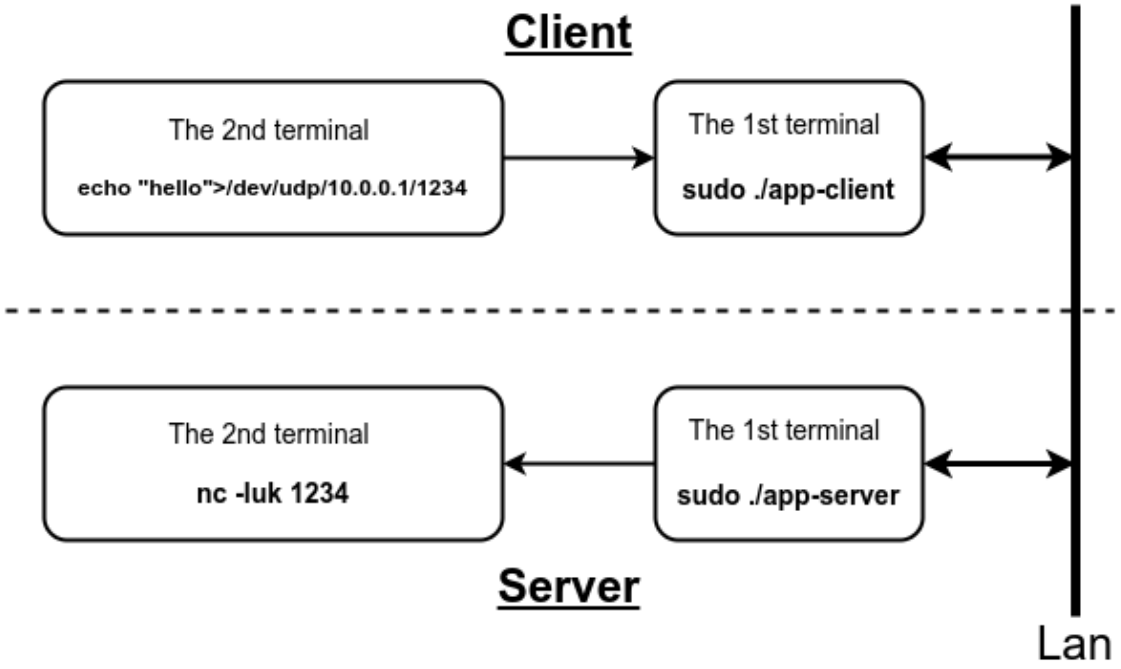

图9：测试示意图

- 在服务器第一个终端上启动服务器端程序 `sudo ./app-server`
- 在服务器第二个终端上执行命令 `nc -luk 1234`，这个命令将一直监听在UDP的1234端口上；
- 在客户端第一个终端上启动客户端程序 `sudo ./app-client`
- 在客户端第二个终端上执行命令 `echo "hello" > /dev/udp/10.0.0.1/1234`，这个命令将向10.0.0.1(服务器端的tun0绑定的IP)的UDP端口1234发送一条消息
- 客户端第二个终端上向10.0.0.1:1234发送了一个UDP消息，内容是：hello
- 最终在服务器端的第二个终端上收到了这个信息
- 下面是运行截图

```
whowin@whowin-ubuntu:~/myTesting/net/tun-tap$
whowin@whowin-ubuntu:~/myTesting/net/tun-tap$ sudo ./app-server
Successfully connected to interface tun0
Successfully bind an address on sock_fd.
Server listening..

Accept the connection request from client.
Server received message from client.
Server received message from client.
Server received message from client.
Server received message from client.
```

图10：服务器端第一个终端

```
whowin@whowin-T430:~$
whowin@whowin-T430:~$ sudo ./app-client
Successfully connected to interface tun0
CLIENT: Connected to server 192.168.2.114
Received data from tun.
Received data from tun.
Received data from tun.
Received data from tun.
```

图11：客户端第一个终端

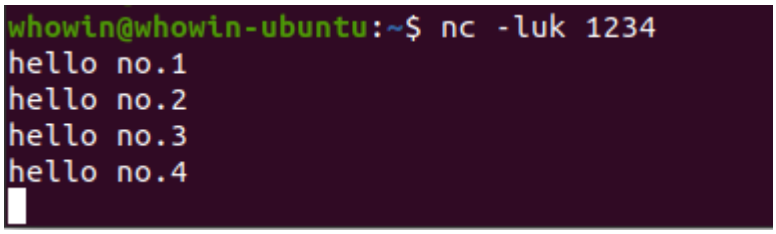

图12：服务器端第二个终端

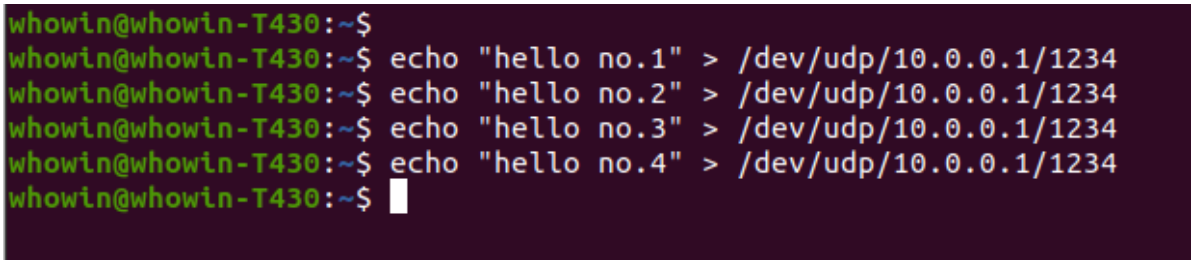

图13：客户端第二个终端

5. 后记

- 我们实现了一个简单的IP隧道，在这个IP隧道，我们传送一个UDP报文，我们传了一个UDP报文而不是一个TCP报文是为了省去connect()的麻烦；
- 这样一个IP隧道并不局限在局域网中，通过互联网一样可以建立一个IP隧道；
- 我们的这个服务器端的程序仅处理了一个客户端的连接，如果我们允许多个客户端接入并建立多条IP隧道，如果连接的多个客户端的tun都绑定在同一个网段上，那么通过服务器显然是可以像局域网一样相互通信的，好像多个终端在一个局域网里一样，是不是有点像 VPN ，实际上很多 VPN 就是使用IP隧道实现的；
- IP隧道还可以用于很多场合，如果你的防火墙不允许某些协议通过，那么你可以通过一个防火墙允许的端口与服务器建立一个IP隧道，然后在这个IP隧道里跑那个不被防火墙允许的协议，就像我们在IP隧道里跑UDP协议一样；
- 建立隧道也不一定非得使用TCP/IP协议，比如可以使用ICMP协议建立一个ICMP隧道，当你的电脑只能ping通你的服务器，其它的所有协议都无法通过防火墙的情况下，使用ICMP协议建立一个ICMP隧道，然后可以在这个隧道里跑其它协议；
- 虚拟网络接口的用途很多，现在的虚拟机、容器等大多使用了虚拟网络接口，希望这篇文章可以让你对虚拟网络接口有个初步的认识。

---

email: hengch@163.com