# 在用户空间中对网络数据加密方案预研

## 一.需求背景

用户的所有数据在网络上传输时都必须是经过加密的，以防止其他人窃取用户隐私信息，因此需要一种方案能够加密所有的网络数据。我们目前是采取IPSec方案，在内核模块中添加数据加密功能(我也没有看过现在的IPSec实现方案)，但是随着客户的增多，客户采用的内核版本也各不相同，我们需要及时将内核加密方案适配到用户版本的内核,因此存在大量的重复性工作。本次预研方案针对这一需求研究了一种在用户空间加密网络数据的方案，该方案将加密功能和内核版本解耦，实现一套加密方案适配所有内核。

## 方案设计概述

方案采用OPENVPN这一开源方案为原型，在其基础上改造加密算法来适应客户的要求。下面将通过原理阐述和实例来解释该方案设计原理。

### 系统分析

如图１所示为Android系统中的网络整体模型，Application代表所有类型的应用，包括常见的Android应用(.apk),Native应用(动态链接程序,静态链接程序)，泛指所有的能够进行网络通信的对象；glibc/bionic代表系统库层，在Android上目前体现为bionic库;daemon代表具体在用户空间实现加密、解密功能的守护进程，还负责读写tun/tap设备；Protocol stack代表内核空间中传输层，目前普遍指TCP/UDP协议;Network stack代表网络层，目前指IP层；physical device是指Android设备上的物理网卡，包括以太网卡,无线网卡，usb接口网卡等；tun/tap是指在内核层中虚拟出来的一个网卡，不具备在物理网络中通信的能力，在设计中主要实现隧道的功能。

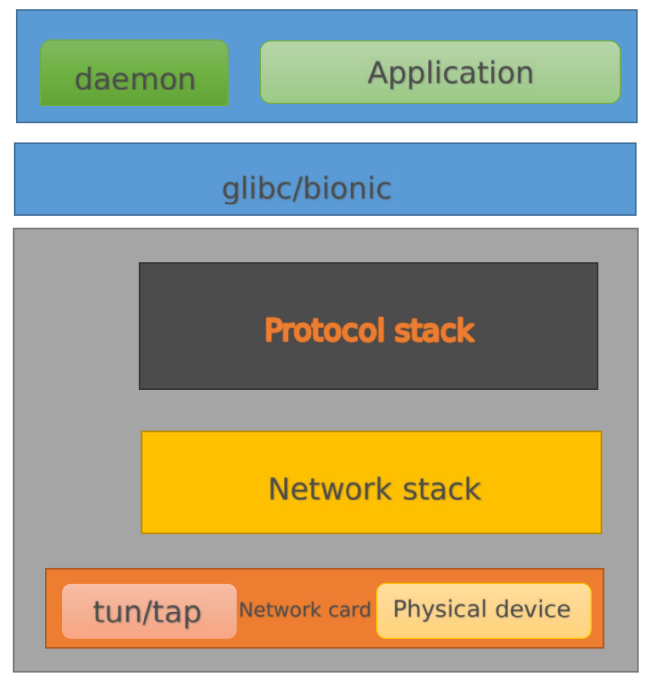


　　　　　　　　　　　　　　　　　图１

## 2.可选方案

应用层发送数据到网络上，我们需要截获所有的网络数据，通常意义上我们能够选择的点为:

1. **hook系统库**

Android系统的可执行应用可分为:apk应用，Native层(C/C++)应用,动态链接和静态链接的二进制可执行文件；除了静态链接之外的所有应用都会经过系统库；

Android中网络访问限制：只有inet，net\_raw，net\_admin，root组才能进行网络通讯，除此之外内核会拒绝创建套接字和网络通讯请求。

Apk应用通信权限:通过在AndroidManifest.xml中配置请求网络权限<android.permission.INTERNET>,系统会将具有权限的应用加入到对应的网络组中，完成赋予应用访问网络权限。

Native程序访问网络权限:vendor通过在init.rc等配置文件中静态添加到网络组中，完成添加权限动作。

动态链接程序,静态链接程序和Native程序类似;

满足以下约束时hook bionic是可行的:

1.用户没有获取root权限的途径

2.vendor合理分配权限,不赋予不明程序root权限或网络权限

从图１中可以看到所有的应用都会经过bionic库，因此hook系统库可以作为一种方案。

以5.1为例中sendto系统调用如何添加hook:

bionic/libc/include/sys/socket.h

\_\_socketcall ssize\_t sendto(int, const void\*, size\_t, int, const struct sockaddr\*, socklen\_t);

更改为:

ssize\_t sendto(int, const void\*, size\_t, int, const struct sockaddr\*, socklen\_t){

Encrypt(void \*,size\_t); //加密处理程序

E\_sendto(int, const void\*, size\_t, int, const struct sockaddr\*, socklen\_t);

}

bionic/libc/arch-arm/syscalls/sendto.S

ENTRY(E\_sendto)

......

ldr r7, =\_\_NR\_sendto

......

END(sendto)

需要hook的网络套接字相关的系统调用:

int recvmmsg(int, struct mmsghdr\*, unsigned int, int, const struct timespec\*);

int recvmsg(int, struct msghdr\*, int);

int sendmmsg(int, const struct mmsghdr\*, unsigned int, int);

int sendmsg(int, const struct msghdr\*, int);

ssize\_t send(int, const void\*, size\_t, int);

ssize\_t recv(int, void\*, size\_t, int);

ssize\_t sendto(int, const void\*, size\_t, int, const struct sockaddr\*, socklen\_t);

ssize\_t recvfrom(int, void\*, size\_t, int, const struct sockaddr\*, socklen\_t\*);

1. **内核系统调用入口加密**

系统调用入口处进行加解密工作，原理是和Android系统库中hook原理是相同的。相比之下不仅适用于android，还适用于所有的linux系统；不仅适用于动态链接文件，也适用于静态链接文件。系统接口层相比内核其他层次来说比较稳定，不轻易增减接口，利于不同版本间的维护。

kernel/net/socket.c

SYSCALL\_DEFINE6(sendto, int, fd, void \_\_user \*, buff, size\_t, len,

unsigned int, flags, struct sockaddr \_\_user \*, addr,

int, addr\_len)

{

　　.....

if (addr) {

err = move\_addr\_to\_kernel(addr, addr\_len, &address);

//加密

}

err = sock\_sendmsg(sock, &msg, len);

}

需要hook的系统调用和上一节是一样的。

1. **虚拟网卡**

这是目前最流行的数据加密方案,利用虚拟网卡的特性来获取所有的网络数据从而完成加解密工作。工作原理见附录A。

1. **结论**

经过调研，虚拟网卡方案最合适:

1. 有成熟的方案可以应用，目前是OPENVPN实现地最好,能够为我们节省大量的工作。
2. 扩展性好，在用户空间中能够快速地替换加密算法。
3. 适配性好，主线内核中提供了tun/tap设备并且提供了稳定的接口，能够适配几乎所有的内核版本
4. 不会对本地进程间通信数据进行额外处理

## ３.原理分析

虚拟网卡方案中如何截获到所有的用户数据:

内核network stack选择路由路径的优先级:主机地址匹配>网络地址匹配>默认路由，而每当一个网络设备初始化时系统自动为接口创建一个直接路由，即和网卡在同一网段的网络地址从该接口向外发;

该方案适用情景分为两种情况:

<1>应用程序访问的服务端ip和本地物理网卡不处于同一网段

服务器和Android端配合实现了隧道功能,Android端和服务端各有两个网卡:物理网卡和虚拟网卡，通过配置实现C/S虚拟网卡处于同一局域网，C/S物理网卡可以直接或间接路由连通。

应用发送的目标地址是在路由时使用默认路由规则到虚拟网卡，从而能够截获到所有发送的数据。Daemon程序获取到所有数据,加密之后发送到物理网路上，从而保证物理网络上数据处于加密状态。

接收数据时,物理网卡接收到的数据首先交给Daemon程序处理，解密之后重新返回给协议栈然后再路由给Application处理。

Daemon程序发送加密完的数据到内核层中，Protocol stack会维护一张socket和端口对应的表，当数据从服务器端返回时根据端口就能够返回给Daemon程序。

<2>应用程序访问的服务端ip和本地物理网卡处于同一网段

和第一种情况不同,根据ip选路的规则，此时应用程序访问服务端的数据包应该会直接从本地物理网卡端口发出,而不会路由到tun设备。

以下方法有一个前提:和物理网卡同网段的服务IP只有一个,Daemon程序和应用已经协商好服务IP；

服务器和Android端配合实现了隧道功能,Android端有两个网卡:物理网卡和虚拟网卡，通过配置实现虚拟网卡作为默认路由。

应用中的目的地址是和物理网卡同网段的IP地址，此时物理网卡的路由匹配”网络地址匹配”规则,不会通过默认路由被路由到虚拟网卡上。这里在netfilter OUTPUT阶段，根据IP和端口号在nat表中做DNAT转换到和虚拟网卡同一网段的地址IPm。接下来的路由阶段被发送到虚拟网卡中，从而能够截获到所有发送的数据。

Daemon程序收到经过DNAT的数据包后重新封装，目的ip是服务器IP,端口号为服务端解密的Daemon程序。

接收数据时，同样首先交给Daemon程序处理，然后重新发送到协议栈中。不同的是需要根据先做SNAT,将IPm转换为服务IP。然后再路由给Application处理。

### ４.示例

<1>应用程序访问的服务端ip和本地物理网卡不处于同一网段

这里建议Android客户端和服务器端配置对称，更容易实现对称隧道的功能。

1.环境配置

Android端网络配置:

tun ip:10.8.0.2/24

eth0 ip:10.0.0.7/24

route:Route add default gw 10.8.0.2 network 0.0.0.0

Iptables:iptable -t nat -A OUTPUT -p any !--dports 23456 -j SNAT --to-source 10.0.0.7

server端网络配置

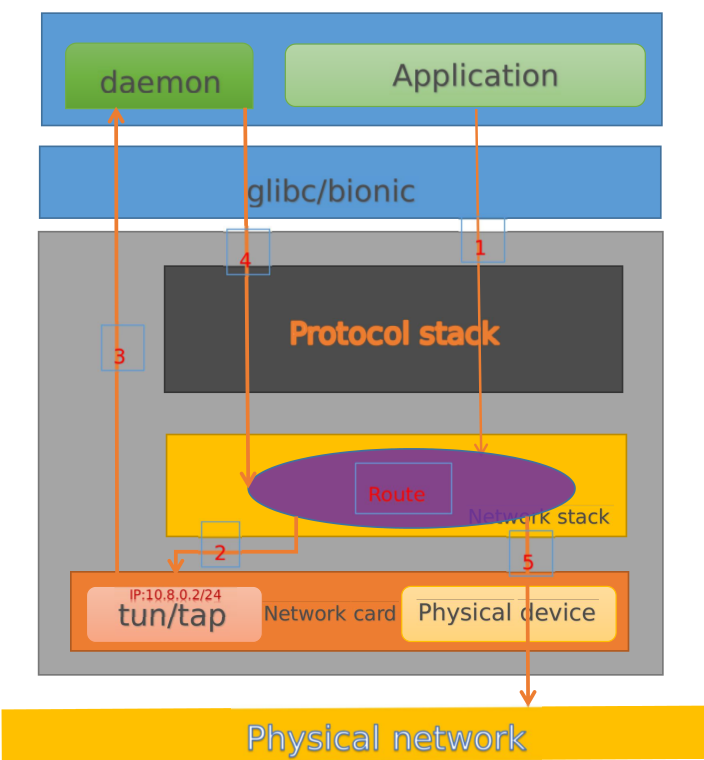
tun ip:10.8.0.3/24

eth ip:10.0.0.6/24

服务端Daemon程序监听端口:23456

以Android客户端应用访问IP为61.223.111.2的服务为例:

发送过程如下:



1. 应用通过套接字发送目的地址为61.223.111.2的数据到内核网络栈中后
2. 通过默认路由到tun设备,tun设备实现了poll接口,接收到消息之后通过poll机制通知Daemon
3. Daemon程序接收到通知并读取tun设备中的数据，并加密
4. Daemon程序加密数据包并重新通过套接字发送目的地址为10.0.0.6,端口号为23456的数据包到内核协议栈
5. 符合IP路由选择条件2，通过物理网卡发送到网络上

数据包格式

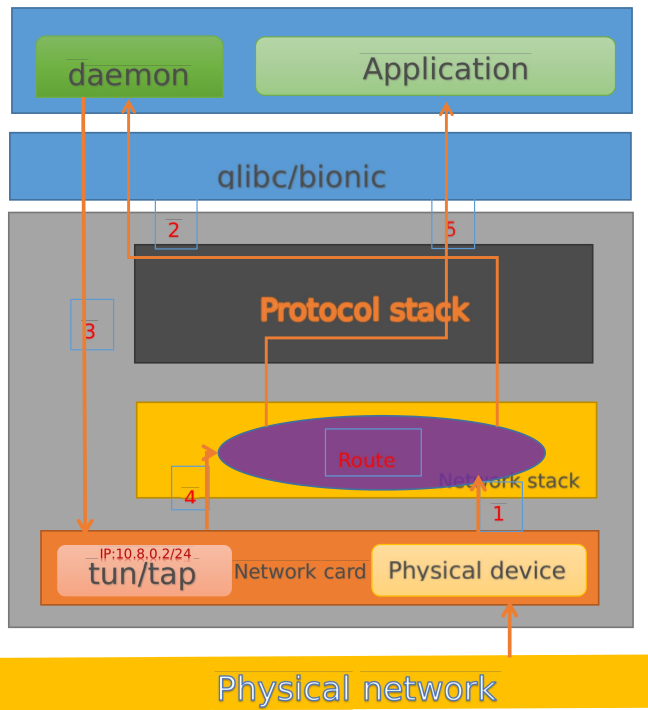
路由到tun网卡上的数据:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 目的地址:61.223.111.2 | 源地址:10.8.0.2 | 目的端口:11111 | 源端口:12345 | 数据 |

发送到物理网络上的数据:

|  |  |  |  |
| --- | --- | --- | --- |
| 目的地址:10.0.0.6 | 源地址:10.0.0.7 | 目的端口:23456 | 源端口:23455 |
| 加密数据 | | | |

接收过程如下:



1. 物理网卡接口接收到数据包
2. 根据目的端口23455，选择daemon程序来处理其socket数据
3. Daemon程序解密数据包,并发送给tun设备
4. Tun设备将数据包发送给协议栈
5. 传输层通过目的端口12345,将返回的数据发送给对应的Application

数据包格式

物理网卡接收到的数据:

|  |  |  |  |
| --- | --- | --- | --- |
| 目的地址:10.0.0.7 | 源地址:10.0.0.6 | 目的端口:23455 | 源端口:23456 |
| 加密数据 | | | |

Daemon程序解密完发送给tun设备的数据:

|  |  |  |  |
| --- | --- | --- | --- |
| 目的地址:61.223.111.2 | 源地址:10.8.0.2 | 目的端口:12345 | 源端口:11111 |
| 解密数据 | | | |

<2>应用程序访问的服务端ip和本地物理网卡处于同一网段

1.环境配置

Android端网络配置:

tun ip:10.8.0.2/24

eth0 ip:10.0.0.7/24

route:Route add default gw 10.8.0.2 network 0.0.0.0

Iptables:

iptable -t nat -A OUTPUT -d 10.0.0.6 -p all !--dports 23456 -j DNAT --to-destination 10.8.0.3

iptable -t nat -A PREROUTING -s 10.8.0.3 -j SNAT --to 10.0.0.7

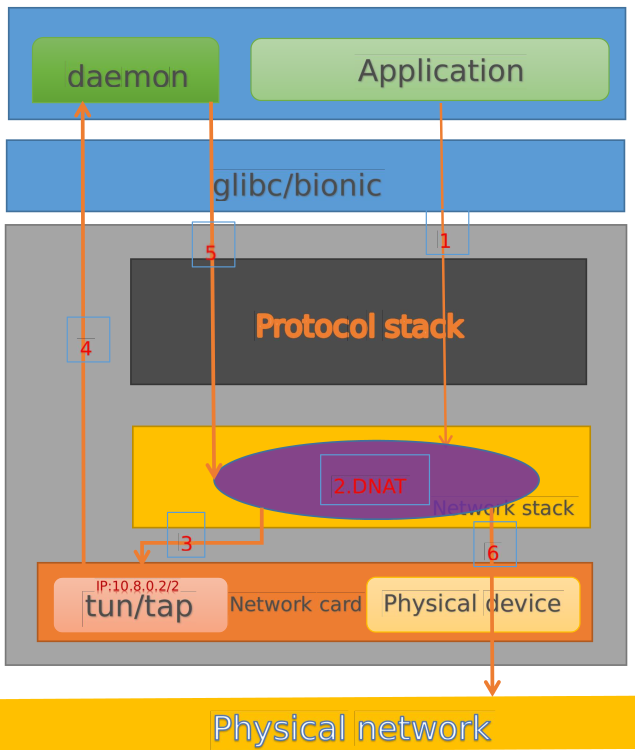
server端网络配置

eth ip:10.0.0.6/24

服务端Daemon程序监听端口:23456

以Android客户端应用访问IP为10.0.0.6,端口号为11111的服务为例:

发送过程如下:



1.应用通过套接字发送目的地址为10.0.0.6的数据到内核网络栈中后

2.在OUTPUT阶段过滤包:目的地址10.0.0.6,目的端口不是23456的包，执行DNAT动作

3.通过默认路由到tun设备

４.Daemon程序一直等待tun中数据,此时读取tun设备中的数据

５.Daemon程序加密数据包并重新通过套接字发送目的地址为10.0.0.6,端口号为23456的数据包到内核协议栈

1. 物理网卡符合路由条件，发送到网络上

数据包格式

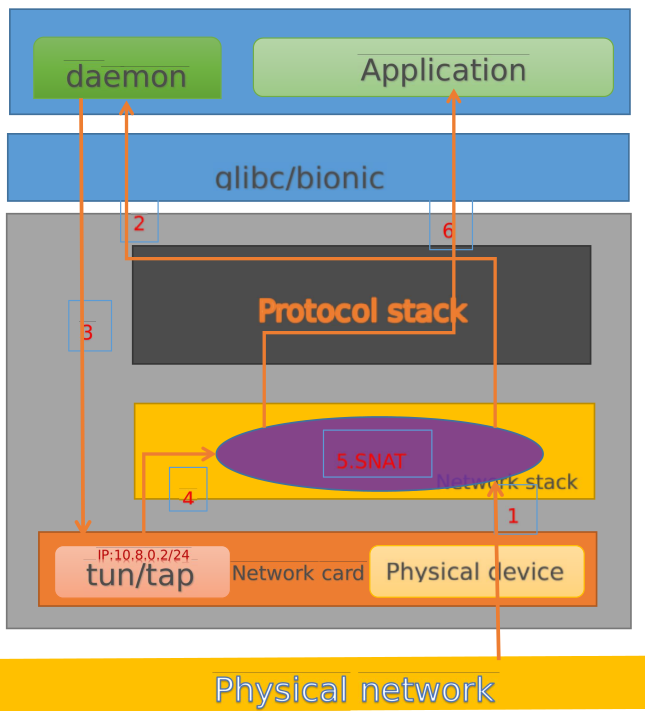
路由到tun网卡上的数据:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 目的地址:10.8.0.3 | 源地址:10.8.0.2 | 目的端口:11111 | 源端口:12345 | 数据 |

发送到物理网络上的数据:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 目的地址:10.0.0.6 | | 源地址:10.0.0.7 | 目的端口:23456 | | | 源端口:23455 |
| 加密数据 | | | | | | |
| 目的地址:10.8.0.3 | 源地址:10.8.0.2 | | | 目的端口:11111 | 源端口:12345 | |

接收过程如下:



1.物理网卡接口接收到数据包

2.直接向上给传输层,根据端口分发给Daemon程序

3.Daemon程序解密数据

4.Daemon通过tun设备将数据返回给协议栈

5.netfilter对于源地址为10.8.0.3的包在PREROUTING中做SNAT

6.传输层通过目的端口12345,将返回的数据发送给对应的Application

数据包格式

物理网卡接收到的数据:

|  |  |  |  |
| --- | --- | --- | --- |
| 目的地址:10.0.0.7 | 源地址:10.0.0.6 | 目的端口:23455 | 源端口:23456 |
| 加密数据 | | | |

Daemon程序解密完发送给tun设备的数据:

|  |  |  |  |
| --- | --- | --- | --- |
| 目的地址:10.8.0.2 | 源地址:10.8.0.3 | 目的端口:12345 | 源端口:11111 |
| 解密数据 | | | |

Netfilter过滤到源地址为10.8.0.3做SNAT动作，之后的数据为:

|  |  |  |  |
| --- | --- | --- | --- |
| 目的地址:10.8.0.2 | 源地址:10.0.0.6 | 目的端口:12345 | 源端口:11111 |
| 解密数据 | | | |

# 附录A tun设备工作原理

### 1.tun设备背景

当初为什么设计出来tun设备已经不可考了,我们只知道其作为一个虚拟网络设备

为用户空间程序提供了网络数据包的发送和接收能力。

首先我们看一下Linux内核网络的分层模型:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 用户空间 | 应用程序 | | | | |
|  | C标准库(glibc) | | |  | tun字符设备接口 |
| 内核应用层 | struct socket  struct sock | | |
| 传输层 | struct proto(TCP/UDP) | | |
| 网络层 | struct packet\_type(IP) | | 特定于协议部分 |
| 网络设备层 | struct net\_device | | |
| 驱动层 | 特定于硬件的驱动 | tun设备 | | |  |

图

上面的分层模型中看,内核应用层负责套接字缓冲区的管理，传输层实现TCP/UDP协议规范,网络层实现IP协议(还有其他的协议，暂时不提),网络设备层向网络层提供了一个抽象接口，隔离了特定的硬件驱动,向下为具体的驱动注册成为标准网络设备的接口,驱动需要实现标准网络接口的实际内容。

上面的模型中有一个网络设备层，作为驱动层和网络层之间的中间层，抽象出网络设备的功能，协议栈不需要关心设备怎么发出数据,而设备也不需要了解协议栈怎么处理.tun就是通过网络设备层提供的接口注册成为了一个标准的网卡设备,同时又作为字符设备注册到了内核中，因此tun设备同时具有这两方面特性:作为网络设备出现在网络模型中，从用户角度看是一个完整的网卡设备,可以正常接收从协议栈发送过来的包,只不过没有真正操纵物理设备发送到网络上;同时又作为一个字符设备注册到系统中,向用户空间提供设备接口来操作设备内的数据。

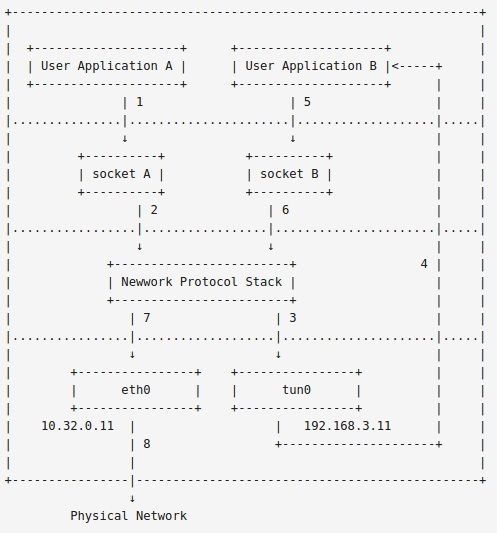
## tun设备的收发过程

tun作为一个三层设备,具有IP地址，所以可以进行路由。

可以将tun设备看做一个真实的网卡,接收数据表示应用程序通过字符设备接口向tun设备中写数据,tun设备接收到数据之后触发中断,告诉网络层来接收数据;网络层接收到数据之后开始层层解析最后报告用户程序有相应的网络包到达,通过socket发送数据给用户程序;

发送数据表示应用程序通过套接字发送数据,经过内核应用层,传输层，网络层的处理,被路由到tun设备,其接收到数据之后具体怎么处理就看应用程序读取tun设备之后如何处理了。

示例:

在用户空间中对网络数据加密方案预研

1. 设置网卡的IP地址:eth0IP:10.32.0.11，tun0IP:192.168.3.11
2. 设置默认路由:route add default gw 192.168.3.11 netmask 0.0.0.0

发送数据过程:

1.应用程序A是一个普通的程序，通过socket A发送了一个数据包，假设这个数据包的目的IP地址是192.168.3.1

2.socket将这个数据包丢给协议栈

3.协议栈根据数据包的目的IP地址，匹配本地路由规则，知道这个数据包应该由tun0出去，于是将数据包交给tun0

4.tun0收到数据包之后，发现另一端被进程B打开了，于是将数据包丢给了进程B

5.进程B收到数据包之后，做一些跟业务相关的处理，然后构造一个新的数据包，将原来的数据包嵌入在新的数据包中，最后通过socket B将数据包转发出去，这时候新数据包的源地址变成了eth0的地址，而目的IP地址变成了一个其它的地址，比如是10.33.0.1.

6.socket B将数据包丢给协议栈

7.协议栈根据本地路由，发现这个数据包应该要通过eth0发送出去，于是将数据包交给eth0

8.eth0通过物理网络将数据包发送出去

接收数据过程:

10.33.0.1收到数据包之后，会打开数据包，读取里面的原始数据包，并转发给本地的192.168.3.1，然后等收到192.168.3.1的应答后，再构造新的应答包，并将原始应答包封装在里面，再由原路径返回给应用程序B，应用程序B取出里面的原始应答包，最后返回给应用程序A

参考:

tun/tap设备:https://segmentfault.com/a/1190000009249039