4over6 实验报告

李文凯　2016011369　计 65
王　琛　2016011360　计 65

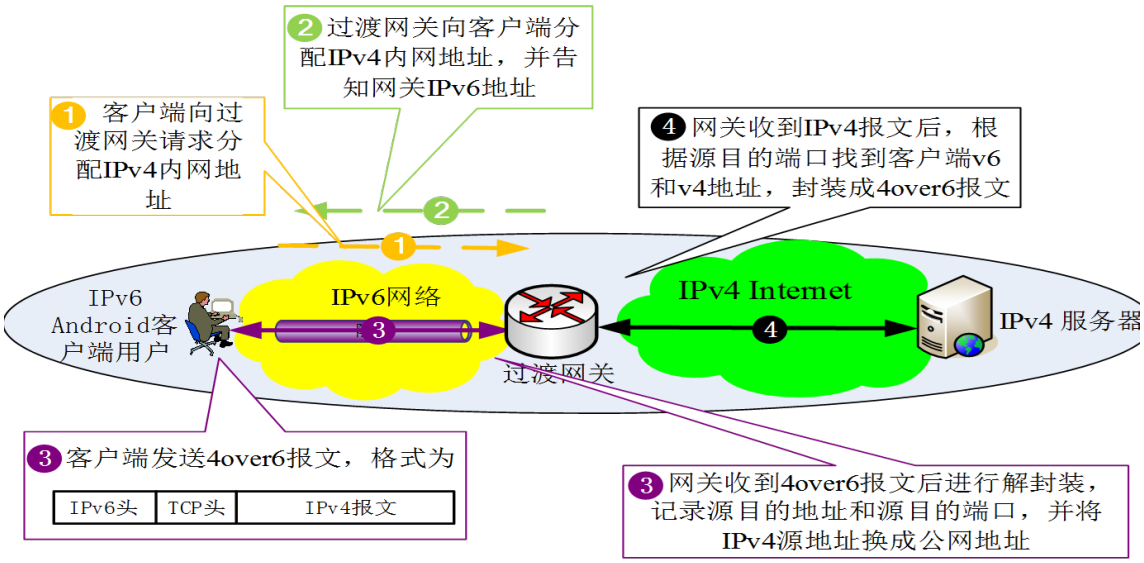
June 17, 2019

1　实验目的

- 掌握 Android 下应用程序开发环境的搭建和使用
- 掌握 IPv4 over IPv6 隧道的工作原理

2　实验原理

4over6 隧道模式是指，将 ipv4 报文作为载荷填入 ipv6 报文中，Android 终端发出的报文由过渡网关负责解封并将 ipv4 报文发送到公网，同时过渡网关将受到的 ipv4 网络数据加以封装，发给 Android 终端。



从图中可以看出，安卓客户端首先向过渡网关分配 IPv4 内网地址，然后过渡网关再提供 IPv4 地址。然后，安卓客户端发送 4over6 协议报文，过渡网关接受报文并进行分析，得到源地址和目的地址，将 IPv4 地址转化为公网地址，发送到公网中。当过渡网关收到

公网的 IPv4 报文之后，根据记录好的映射关系，重新封装成 4over6 报文，发给对应的客户端，从而完成数据的转发和接受。

在以上流程中，用户处于纯 IPv6 网络环境之中，过渡网关运行双栈协议。

3 实验设计与内容

3.1 实验分工

本次实验李文凯同学负责客户端，王琛同学负责客户端的流量显示部分以及服务器端。

3.2 客户端

3.2.1 功能划分

客户端程序主要分成两个模块，前台由 Android 编写，主要是接受用户的输入和相关指令，启用 VPNService，并显示流量统计信息，而后端由 C++ 编写，负责网络数据收发。二者调用基于 JNI 实现，数据交互通过管道实现。

3.2.2 主要技术

JNI 与 NDK 对于网络相关的操作，往往在字节层面，并且对于内存管理的要求也更高，使用 Java 不便，因此还是采用了惯用的 C/C++ 进行网络数据交互处理，由此需要在基于 Java 的 Android 开发工作中引入 C/C++ 模块，所以我们需要使用 JNI 和 NDK。

JNI（Java Native Interface）提供了若干的 API 实现了 Java 和其他语言的通信，使得我们在本实验中能够实现本地方法 native method，并在 Java 程序中调用他们。而 NDK(Native Development Kit) 则是方便开发者将 C/C++ 程序打包并加入到 APK 中，供 Android 程序使用。

当前最新的 Android Studio 已经提供了完善的 JNI 支持，相比之前，省去了编译头文件等复杂操作，框架部分已经由内置模板实现，十分方便。

VPNService

- 应用程序使用 socket，将相应的数据包发送到真实的网络设备上。一般移动设备只有无线网卡，因此是发送到真实的 WiFi 设备上；

- Android 系统通过 iptables，使用 NAT，将所有的数据包转发到 TUN 虚拟网络设备上去，端口是 tun0；
- VPN 程序通过打开/dev/tun 设备，并读取该设备上的数据，可以获得所有转发到 TUN 虚拟网络设备上的 IP 包。因为设备上的所有 IP 包都会被 NAT 转成原地址是 tun0 端口发送的，所以也就是说你的 VPN 程序可以获得进出该设备的几乎所有的数据（也有例外，不是全部，比如回环数据就无法获得）；
- VPN 数据可以做一些处理，然后将处理过后的数据包，通过真实的网络设备发送出去。为了防止发送的数据包再被转到 TUN 虚拟网络设备上，VPN 程序所使用的 socket 必须先被明确绑定到真实的网络设备上去。

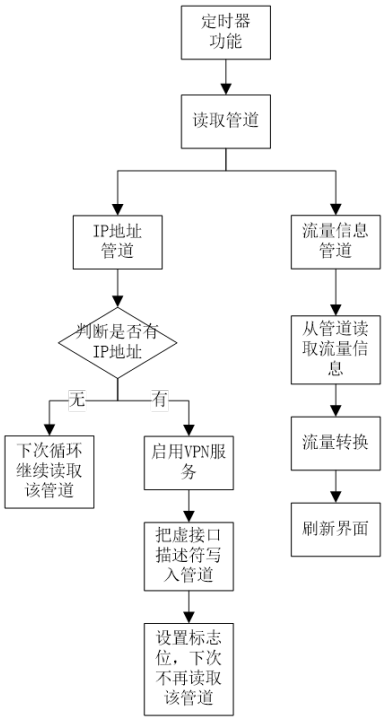
在实际使用中，需要将 Android 设备与过渡网关相联的 v6 socket 保护起来，其他连接则纳入 VPNService 管辖范围。

管道 管道其实就是特殊的文件，不同语言下数据交互可以就可以通过文件读写操作来完成。有两个地方需要注意：

- 管道是持久化存储在设备外存的，如果程序退出未进行清理，下次运行程序仍会读取到上次的残留信息，可能引发错误。
- 双向读写 or 单向读写？本次实验为了简化操作逻辑，每个管道都是一方可读另一方可写，否则需要额外的判断或同步机制。

3.2.3 整体工作流程

3.2.4 前台



前台的工作总结为以下几个：

- 读取用户输入的过渡网关 ipv6_address, port，传给后台线程用来发出连接请求
- 接受后台获得的来自过渡网关的连接响应，用来启用 VPNService，并将生成的 tun0 描述符传给后台，供其进行数据交互

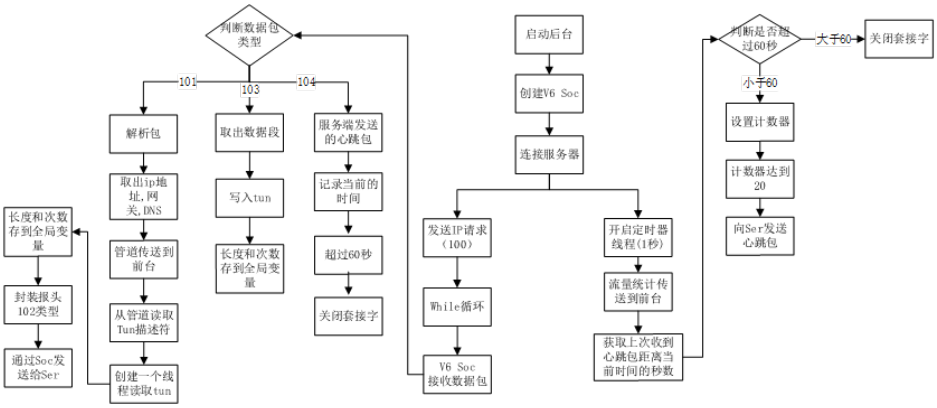
- 定时获取来自后台的流量统计信息，并显示。

实现要点：

- backend 后台线程：在点击连接按钮后，创建线程运行 native method 执行后台线程
- 定时服务：由 Java 提供的 TimerTask 实现
- VPNService：继承 Android 提供的 VpnService 实现

3.2.5 后台

后台工作流程如图所示



后台主要有以下几个重要的工作线程：

- 读取 ip_pipe：获取 VPNService 创建的虚接口，以及程序退出信号
- 读取虚接口 tun0：Android 设备所有的上网请求都会发送到 tun0，该线程负责对其进行封装，发送到过渡网关
- 读取 ipv6 socket：获取来自过渡网关的数据，分类别进行处理
- 计时器：保活机制，发送流量统计信息到前台

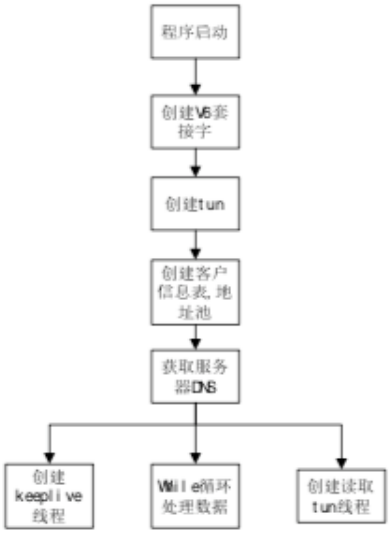
实现要点：

后台主要是 C/C++ 的处理逻辑，除了基础的多线程处理之外没有其他需要额外注意的地方。但是在现有的实现框架下，多个线程对一个 socket 进行读写操作，经过调研，Linux 下的 send 函数并不是原子操作，因此可能出现发送心跳包的同时，发送上网请求包，导致混包的情况出现，因此必须给 socket 的读写操作加锁，防止该情形的发生。

3.3 服务器

3.3.1 工作流程

服务器的总体工作流程如图：



主要分为如下三个部分：

1. 主进程接受客户端连接，接受不同类型的数据包。

2. 创建 keepalive 线程，给每个正在连接的客户端发送心跳包。

3. 读取虚接口，包括将数据发送出去，以及接受虚接口数据然后发送给客户端。

3.3.2 具体实现

具体的客户端实现我们采用了近年来比较流行的 epoll 模型，用来代替实验指导书的 select 模型。使用 epoll 模型能够进行 I/O 多路复用，只用单线程即可完成任务（keepalive 仍然需要新的线程）。并且 epoll 模型不像 select 模型有最大用户数目的限制，因此有了广泛的使用。

使用 epoll，遍历所有的事件，可以方便判断是新的用户请求连接，还是接收到了用户数据，或者接收到了虚接口 tun 的数据，再做相应的处理。监听所有客户端的 socket，当其中一个 socket 可以读或者可以写时，首先找到 socket 对应的用户，然后根据类型做处理。如果 socket 可写，则从该用户对应的数据队列中取出数据然后写入 socket。如果 socket 可读，则读取消息，然后写到 tun。tun 同样注册入 epoll。如果 tun 接收到了数据，那么进行将其转发给客户端。代码框架如下，省略了一些细节的处理。

```
while(true) {
    nfds = epoll_wait(epfd, events, 20, 500);
    for(int i = 0; i < nfds; i++) {
        if(events[i].data.fd == listenfd) { //listen event
            printf("listening event\n");
            connfd = accept(listenfd, (struct sockaddr *)&clientaddr, &client_len);
            client_fd = connfd;
            if (connfd == -1) {
                perror("accept failed");
                close(connfd);
                exit(-1);
            }
            printf("client fd: %d\n", client_fd);

            int i = 0;
            pthread_mutex_lock(&mutex);
            for (; i < MAX_USER; i++) {
                if (user_info_table[i].fd == -1) {
                    user_info_table[i].fd = connfd;
                    memcpy(&(user_info_table[i].v6addr), &clientaddr, sizeof(struct
                        sockaddr));
                    user_info_table[i].secs = time(NULL);
                    user_info_table[i].count = 5;
                    break;
                }
            }
            pthread_mutex_unlock(&mutex);

            //accept connection
        } else {
            if(events[i].data.fd == tun_fd) {
                process_packet_to_tun(client_fd);
            } else if(events[i].events & EPOLLIN) {
                //receive from user
            }
        }
    }
}
```

开始运行时，首先要打开服务器的监听 socket，具体实现的 IPv4 的情况十分类似，只是需要将其修改为 IPv6 情形下。设置完监听之后，将其放入 epoll 中。

```
struct sockaddr_in6 server_addr;
server_addr.sin6_family = AF_INET6;
server_addr.sin6_addr = in6addr_any;
server_addr.sin6_port = htons(SERVER_PORT);
int ret = bind(listenfd, (struct sockaddr *) &server_addr, sizeof(server_addr));
if (ret == -1) {
    perror("server bind error");
    close(listenfd);
    exit(-1);
}
if( (ret=listen(listenfd, MAX_USER)) < 0 ) {
    perror("Server listen failed");
}
```

用户信息根据实验指导书，保存在 user_info 的结构体中。每个用户分配一个 ip 地址

和一个 socket。使用 keepalive 线程，每隔一秒钟向用户放松心跳包，当某个用户发送的心跳包超时时，则从队列中删除掉该用户。注意 keepalive 线程发送包时需要加上锁，防止写数据出错。

在服务器端，使用 iptables 做 NAT，实现私有网段和公网地址的转换。在 c++ 中可以使用 system 函数运行 shell 命令。iptables 会建立和服务器端口之间的映射，并且将 IP 报文的源地址、源端口转换为服务器的公网 IP 和端口。我们使用 10.0.0.1/24 作为私有网段。相关的一些配置如下：

```
char buffer[256];
sprintf(buffer,"ip␣link␣set␣dev␣%s␣up", ifr.ifr_name);
system(buffer);
sprintf(buffer,"ip␣a␣add␣10.0.0.1/24␣dev␣%s", ifr.ifr_name);
system(buffer);
sprintf(buffer,"ip␣link␣set␣dev␣%s␣mtu␣%u", ifr.ifr_name, 1500 - MSG_HEADER_SIZE);
system(buffer);
```

3.3.3 编译运行

server 文件夹下已经有 makefile，使用 make 命令即可编译，开启了 pthread 和 std=c++11 的编译选项

使用 make run 命令可以运行

3.3.4 运行情况

下图是服务器收到用户包和发送包给用户的情况：

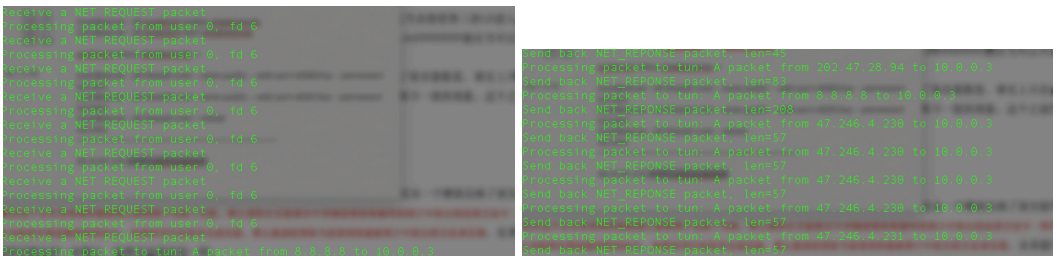


Figure 1: 服务器运行截图

4 实验结果

我们的安卓客户端是华为 Nova，服务器运行在日本的 VPS 主机，提供双栈环境，使用的网络是宿舍楼的 Tsinghua-5G（不进行联网认证）。我们对以下的应用进行了测试：

- 通讯工具：我们对微信和 QQ 进行了测试，发现能够正常接收消息

- 视频应用：我们测试了抖音、bilibili 网站。在服务器环境稳定的情况下，这些应用能够流畅的运行。
- 网页：我们测试了百度、bing、google 等网站，能够流畅加载。

具体的运行情况还可以查看我们提供的运行效果视频。

5 实验中遇到的问题

我认为我们遇到的主要问题是有时候接收到错误的消息。这种情况出现的概率并不高，一般只有程序运行足够长时间后才会出现。经过我们的仔细分析，发现是由于不同线程对 socket 读写造成的，会造成数据错乱，因为 socket 的读写操作并不是原子性的。对此，网上给出的建议是将所有的写操作都统一由一个线程完成，或者使用加锁的方式。但是经过我们调研，业界对于多线程 IO 一个 socket 的做法是不推荐的，对于 tcp 连接来说，这样会严重影响数据传输的效率。我们实验结果显示，加锁之后，网络带宽有所降低，但是之前遇到的问题没有再次发生。这个问题一个比较好的解决思路是设置两个 socket 连接，单独收发，分别设立缓冲区和分隔符，使得收发数据的操作被单独封装起来，一同处理，避免出现线程安全问题。但是由于时间原因，我们没有实现。

在我们参考的相关资料中，大家并没有关注这个问题带来的隐患，如果不做处理，短时间测试不会出现问题的概率很小，但我们通过长达几十分钟乃至几小时小时的视频放映测试中发现了这个问题。

6 实验总结

6.1 实验心得

这次实验是对上学期所学的网络原理知识的综合运用。总体来说，实验的完成还是十分顺利的。通过这个实验，感觉自己对于网络方面的编程水平有了进一步的提高。看到自己编写的程序能够真正流畅运行，成就感还是相当高的。在整个实验完成的过程中，我们学习了安卓 VPN service 框架，掌握了 epoll 模型的运用，提升了服务器的性能。

再次还要感谢老师和助教的指导，没有你们，我们无法这么顺利的完成实验。

6.2 意见和建议

建议可以在服务器中新增一些实用路由器的一些功能，比如数据加密传输等等，提高这个实验的实用性。还有上一部分提到的多线程 IO socket 问题。