

IPV4 OVER IPV6 隧道协议实验报告

计 45 班 谭思楠 2013011720

计 42 班 劉家昌 2014011307

IPv4 over IPV6 隧道协议实验报告

实验目的

实验内容

客户端

服务器端

协议定义

IPv4 数据报

4over6 数据包

实现方法

网络封包处理

IPv4 数据报

4over6 数据包

前后端通信

通信方法

数据包结构

后端回复消息

程序资源

客户端后端

服务器

流程逻辑

客户端后端

后台整体流程

主循环逻辑

失败 / 关闭流程

处理 over6 数据包

处理 over6 写

处理前端信息

服务器
 后台整体流程
 主循环逻辑等
前端实现
实验结果
 运行情况
 前端界面
 正常上网
测试结果
 网络测试结果
 功能测试情况
实验指导书的建议
 TCP...?
 4over6...?
 端序...
 int... char... packed...

实验目的

IPv4 over IPv6，简称「4over6」是 IPv4 向 IPv6 发展进程中，向纯 IPv6 主干网过渡提出的一种新技术，可以最大程度地继承基于 IPv4 网络和应用，实现 IPv4 向 IPv6 平滑的过渡。该实验通过实现 IPv4 over IPv6 隧道最小原型验证系统，对 4over6 隧道的实现原理不能有更加深刻的认识（因为根本就不是 4over6）。

除此之外，在这个实验中还有无数的槽点。

实验内容

客户端

我们在安卓设备上实现一个 4over6 隧道系统的客户端程序，内容如下：

1. 实现安卓界面程序，显示隧道报文收发状态（Java 语言）；
2. 启用安卓 VPN 服务（Java 语言）；
3. 实现底层通信程序，对 4over6 隧道系统控制消息和数据消息的处理（C 语言）。

为了完成本实验的实验要求，我们完成了以下 2 项内容：

1. Java 语言的显示前台界面：

1. 进行网络检测，显示物理接口 IPv6 地址信息；
2. 请求 4over6 服务器信息，启动后台进程连接网络；
3. 定期刷新显示：4over6 网络 IP 信息、包数、总流量与速度信息和连接时长；
4. 开启安卓 VPN 服务，将虚接口 `fd` 传输予后端；
5. 信息通讯由管道完成。

2. C 语言实现的 4over6 客户端：

1. 连接 4over6 服务器；
2. 从管道获取虚接口 `fd`，读写 VPN 网络数据；
3. 将前端所需的 4over6 网络 IP 信息、包数、总流量与速度信息和连接时长传予前端；
4. 对数据进行 4over6 解封装，读写连接的网络数据；
5. 定时给服务器发送 keepalive 消息。

服务器端

由于我组只有两个人，根据课程说明不需要完成服务器端的工作。但是由于课程提供的服务器端经常出现异常，我们实现了一个简易的不支持多终端登录的服务器端；除此之外与实验要求完全相符：

1. 实现服务端与客户端之间控制通道的建立与维护；
2. 实现对客户端网络接口的配置；
3. 实现对 4over6 隧道系统数据报文的封装和解封装。

为了完成本实验的实验要求，我们完成了以下几项功能：

1. 创建 IPv6 TCP 套接字，监听客户端接入；
2. 维护 tun 虚接口，实现对虚接口的读写操作；
3. 读取客户端从 IPv6 TCP 套接字发送来的数据，实现对系统的控制消息和数据消息的处理；
4. 实现对数据消息的解封装，并写入虚接口；
5. 实现对虚接口接收到的数据报文进行封装，通过 IPv6 套接字发送给客户端；
6. 实现保活机制，监测客户端是否在线，并且定时给客户端发送 keepalive 消息。

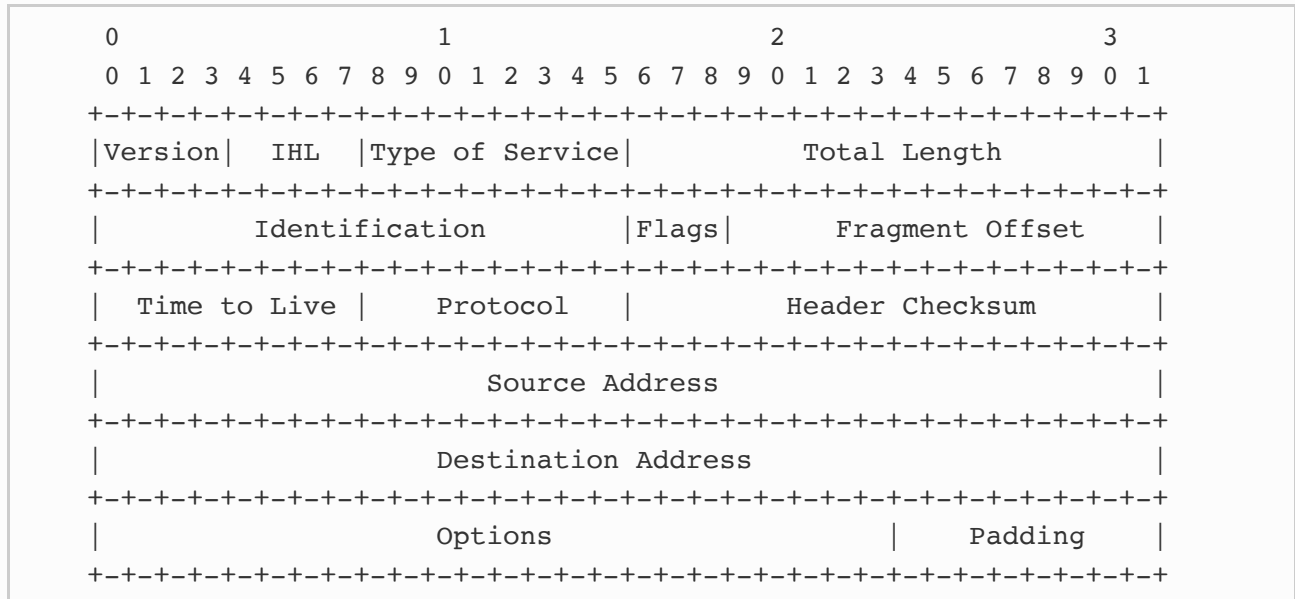
我们认为这应当成为我组的加分项。

协议定义

本实验需要需要在程序层面处理两类数据协议类型：一是从 VPN/tun 虚接口读写数据获得的三层网络层协议封包，亦即 IPv4。二是维护客户端和服务端之间通信的 4over6（然而并不是）七层应用层数据包，下层协议为 TCP。

IPv4 数据报

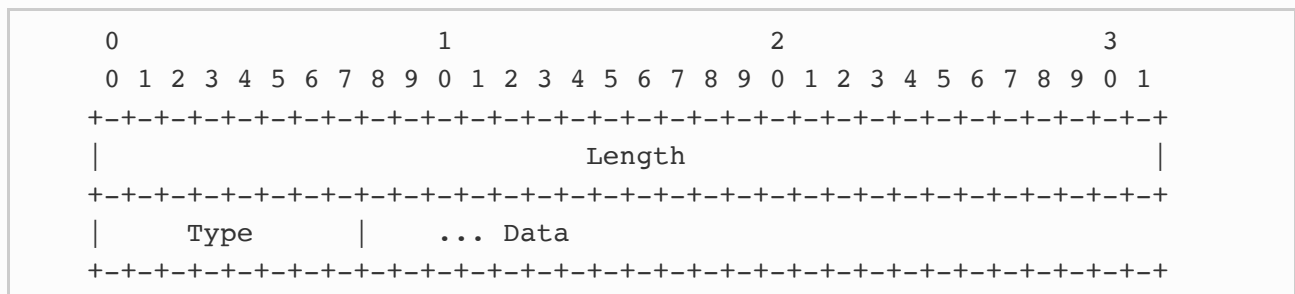
需要处理的 IPv4 报头基本结构如下，来自于 RFC 791 INTERNET PROTOCOL:



而随后的数据，为根据 `Total Length` 指定长度的 IP 封包内的数据。

4over6 数据包

需要处理的 4over6 包头基本结构如下，结构来自于实验指导书：



数据包的行为、长度和内容，根据 Type 不同，分别为：

1. 若 `Type` 为 100，则 `Data` 为空。4over6 包为客户端 IP 地址请求；
2. 若 `Type` 为 101，则 `Data` 为长度为 `Length` 「IP 掩码 DNS1 DNS2 DNS3」格式的字符串数据，以空格分割；
3. 若 `Type` 为 102，则 `Data` 为根据 `Length` 指定长度的 IP 封包数据，但方向是客户端到服务器；
4. 若 `Type` 为 103，则 `Data` 为根据 `Length` 指定长度的 IP 封包数据，但方向是服务器到客户端；
5. 若 `Type` 为 103，则 `Data` 为空。4over6 包为心跳包。

程序定义如下：

```
typedef struct {
    uint32_t length;
    uint8_t type;
    uint8_t data[];
} __attribute__((packed)) over6Packet;
```

需要注意，`uint32_t` 是必须的，否则会由于不同的内存模型出现不同的数据大小，导致包无法正常分析；`__attribute__((packed))` 也是必须的，否则编译器可能 align 其中的 `uint8_t`，与包的实际情况不符。

具体请参见 `app/src/main/cpp/communication.h` 文件。

实现方法

网络封包处理

IPv4 数据报

其中我们注意的仅有：`Total Length`、以及 `Destination Address`。事实上，由于从虚设备中直接 `read` 读取，获得的一定是一个完整的 IP 封包，因此并不需要特别处理 `Total Length`。并且由于我们组根据实验要求不需要实现服务器端，`Destination Address` 的处理也可以忽略不计。

但为了报告完整性：服务器应当在从虚设备读取到 IPv4 数据报时检查数据的目的地址，并且模拟为一个路由器的功能，将这一数据报发往正确的客户端；但是事实上，如果写服务器的同学比较偷懒，可以将实现的程序当成一个集线器，直接将收到的包发往所有的客户端，而因为客户端会再次检查 IP 包的目的地址，不会出现异常的行为。这一做法虽然浪费了资源，但是可能在一定程度上会提升效率（没有了查找表的 overhead）。

除此之外，IPv4 数据报的处理方法与 4over6 基本一致。

但是要特别注意的是与虚设备直接写入不同，由于非阻塞 TCP 的写可能会由于缓冲区满而失败，因此必须将数据报先放入程序自己的缓冲区队列中，等待主循环取出，封装为 4over6 包，发送到客户端/服务器。

具体实现，可以参见 `app/src/main/cpp/communication.cpp` 下，和 `DroidOver6Server.c` 文件的 `rawToOver6` 函数。

4over6 数据包

这里最大的槽点在于，4over6 的通信使用的是 TCP 协议，因此事实上，4over6 数据不能称为 4over6 数据包，而是 4over6 数据流。但是因为实验指导书非要我们实现成包的样子，就需要通过长度段来分包（我已经想象到了一堆人问粘包问题怎么解决的现场）。

而用长度段来分包最大的槽点在于，TCP 并不是完全可靠的协议，因此长度段的传输可能出现错误，那么后面的包就无法正确分割，会对协议带来灾难性的后果。这是拿 TCP 传输「包」的天然恶意，作为课程学生我们不能一任性写了 UDP 可能就拿不到分了，所以只能尽可能解决 TCP 带来的问题。

但是要特别注意的是与虚设备直接读出数据报不同，处理 4over6 的时候，需要特别注意的是单次 read 并不一定可以获得完整的整个「包」，可能包的另外一部分还在流中没有传输过来。因此需要实现一个 buffer，检测 buffer 内的数据是否足够切出一个完整的 4over6 包，再进行处理。

对于客户端而言：

1. 若 `Type` 为 100，则应当丢弃，因为不知道是哪个服务器犯傻把隔壁的请求包发过来了；
2. 若 `Type` 为 101，则应当解析「IP 掩码 DNS1 DNS2 DNS3」格式的字符串数据，发往前端配置 VPN 信息；
3. 若 `Type` 为 102，则应当丢弃，因为还是不知道是哪个服务器把隔壁的请求包发过来了；
4. 若 `Type` 为 103，则应当取出其中的 IP 报文，写入虚设备中；
5. 若 `Type` 为 103，则 4over6 包为心跳包，应当更新超时计时器。

具体实现，可以参见 `app/src/main/cpp/communication.cpp` 下的 `over6Handle` 函数。

对于我组额外实现的服务器而言：

1. 若 `Type` 为 100，则应当向客户端发送「IP 掩码 DNS1 DNS2 DNS3」格式的字符串数据，提供配置信息；由于这一信息在通信开始前建立，因此不必要放入 buffer 中排队，但如果要实现一个中途更新信息的服务器，则需要排队；
2. 若 `Type` 为 101，则应当丢弃，因为不知道是哪个客户端把回复包发回来了；
3. 若 `Type` 为 102，则应当取出其中的 IP 报文，写入虚设备中；
4. 若 `Type` 为 103，则应当丢弃，因为还是不知道是哪个客户端把收到的包发回来了；
5. 若 `Type` 为 103，则 4over6 包为心跳包，应当更新超时计时器。

具体实现，可以参见 `DroidOver6Server.c` 文件的 `over6Handle` 函数。

前后端通信

通信方法

前后端通过一个Linux的管道（pipe）机制进行通讯。在前端启动后台进程之前，会创建两个管道。其中一个管道为命令管道（Command Pipe），另外一个为答复管道（Response Pipe）。管道的文件描述符在线程启动时传递给后端线程。

简单起见，所有的前后端通讯均由前端发起，通过命令管道向后端发送一个命令。后端根据命令

更新自己的状态，并通过答复管道回复自己的状态。

数据包结构

前端向后端发送的命令数据包格式如下所示:

```

0           1           2           3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      Command      |    ... Data    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

```

数据包的行为、长度和内容，根据 Command 不同，分别为：

1. 发送 `BACKEND_IPC_COMMAND_TERMINATE` 表示终止后端运行。
2. 发送 `BACKEND_IPC_COMMAND_STATUS` 表示查询VPN的运行状态（包括正在初始化，正在连接远程服务器，连接已断开等）。
3. 发送 `BACKEND_IPC_COMMAND_STATISTICS` 表示查询服务器的VPN的IP配置信息。这些信息对于完成VPN的建立是非常必要的
4. 发送 `BACKEND_IPC_COMMAND_CONFIGURATION` 表示查询服务器的VPN的IP配置信息。这些信息对于完成VPN的建立是必要的。但是在刚刚连接到服务器时，IP地址的配置信息并没有获得。因此前端会在连接建立之后再请求这些信息。之后在通过Android的接口获取VPN的tun设备的文件描述符。
5. 发送 `BACKEND_IPC_COMMAND_SET_TUNNEL_FD` 用于在前端通过VPN接口获得tun设备的文件描述符之后，将其发送给后端。

在上述命令中，除了BACKEND_IPC_COMMAND_SET_TUNNEL_FD之外，Data均为空。命令为BACKEND_IPC_COMMAND_SET_TUNNEL_FD时，数据包格式如下：

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      Command          |                          TUNFD
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|      TUNFD           |       (END)
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

后端回复消息

1. 如果是 `BACKEND_IPC_COMMAND_TERMINATE`，则终止 VPN，进入「失败／关闭流程」；
2. 如果是 `BACKEND_IPC_COMMAND_STATUS`，则通过 IPC 返回当前的 VPN 状态；
3. 如果是 `BACKEND_IPC_COMMAND_STATISTICS`，则通过 IPC 返回当前的 VPN 数据；
4. 如果是 `BACKEND_IPC_COMMAND_CONFIGURATION`，若是没有获得 IP 信息，则向 over6 服务器发送 IP 请求包；通过 IPC 向前端已获得的 IP 地址，或者请等待信息。

5. 如果是 `BACKEND_IPC_COMMAND_SET_TUNNEL_FD`，则设定 VPN 的虚设备 fd。

具体实现可以参见 `app/src/main/cpp/frontend_ipc.cpp` 文件。

在上述命令中，`BACKEND_IPC_COMMAND_STATUS`，`BACKEND_IPC_COMMAND_STATISTICS` 和 `BACKEND_IPC_COMMAND_CONFIGURATION` 需要答复。

`BACKEND_IPC_COMMAND_STATUS`的回复格式如下：

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	
	Status										(END)																												
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	

一个字节的Status值可能为下列

值：`BACKEND_STATE_CONNECTING`、`BACKEND_STATE_WAITING_FOR_IP_CONFIGURATION`、`BACKEND_STATE_CONNECTED`、`BACKEND_STATE_DISCONNECTED`。分别表示后端正在连接，正在等待IP配置，已连接，或者连接已经断开。

`BACKEND_IPC_COMMAND_STATISTICS`的回复格式如下：

0										1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1								
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							
+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+	-	+							

InBytes、OutBytes为64位的I/O字节数。InPackets、OutPackets为32位的I/O数据包数目。

`BACKEND_IPC_COMMAND_CONFIGURATION`的回复格式如下：

1. 与隧道客户端的 IPv6 TCP 套接字；
2. 运行时创建的操作系统 tun 虚接口；
3. 用于非阻塞请求检查事件的 epoll 文件描述符；
4. 上次接受到心跳包的时间，和上次发送心跳包的时间；
5. 从 4over6 流中读取的数据的 buffer，和要往 4over6 流中写入数据的 buffer。

为了报告完整性，如果我们实现完整版的服务器，则需要客户信息表和地址池信息。

具体分配的资源，可以参见 `DroidOver6Server.c`。

流程逻辑

客户端后端

后台整体流程

1. 创建上述资源，包括连接 4over6 服务器，创建 epoll 文件描述符，初始化心跳数据，初始化统计数据；
2. 将 IPv6 TCP 套接字、与前端通信管道加入到 epoll 事件监测中；
3. 向前端发送通知 VPN 开始的信息；
4. 进入「主循环」——

注意此时虚接口还没有进入 epoll，原因是在 IP 地址获得之前，安卓系统都不会提供虚接口 fd。在主循环中，需要检查是否获得了这一接口，再加入到 epoll 中。

主循环逻辑

由于定时器线程过于丑陋，并且 Android 没有时间处理的优雅的 timerfd，可以对 epoll 进行超时处理，在主循环中直接处理发送心跳包、检查心跳包超时的时间。

1. 以发送心跳包 / 检查心跳包的时间为超时时间，进行阻塞的 `epoll_wait` 等待读写或故障事件发生；
2. 若是检查心跳包超时，意味着服务器可能已经掉线，跳到「失败 / 关闭流程」；
3. 若是发送心跳包超时，则发送心跳包。注意这里只能进行记录，而不能直接发送，否则可能破坏上一个包未结束的流；
4. 检查事件，如果有故障事件，跳到「失败 / 关闭流程」；
5. 检查事件，如果 TCP 套接字可读，则需要将其读到 over6 「读出缓冲区」中；
6. 检查事件，如果虚接口设备可读，则需要将其读出到 over6 「写入缓冲区」中；
7. 检查事件，如果前后端通信管道可读，则需要处理前端发来的信息，即进入「处理前端信息」流程——
8. 检查事件，如果 TCP 套接字可写，则需要从缓冲区中取出信息，写入套接字，即进入「处理 over6 写」流程——
9. 进入「处理 over6 数据包」流程——
10. 若是虚接口已经从前端获得，则将其加入 epoll 中监测状态；

11. 只有在设备有东西可写时，才监测可写事件。

失败／关闭流程

1. 如果隧道客户端的 IPv6 TCP 套接字没有关闭，关闭之；
2. 如果安卓系统提供的 VPN 虚接口设备文件描述符没有关闭，关闭之；
3. 如果前后端通信的管道没有关闭，关闭之；
4. 向前端发送 VPN 关闭的信息。

处理 over6 数据包

1. 检查当前缓冲区的第一个包是否能够处理，即缓冲长度是否长于包，如果不是结束流程；
2. 根据「网络封包处理」部分的逻辑处理包，例如更新心跳包时间，写入虚接口，读取配置等等；
3. 将处理过的包从缓冲区清除。

处理 over6 写

1. 如果有上次没有发送完的数据，继续发送，以 3 的流程进行检测；
2. 如果有心跳包需要发送，则发送心跳包；
3. 取出缓冲区中的数据，封装为 102 包格式，`write` 到虚设备中；检查返回值，如果没有发送完整，则从流程中返回，记录并在下次 over6 写流程的开始继续发送；
4. 从缓冲区中将已经成功发送的数据清除。

处理前端信息

1. 根据「前后端通信」部分「消息类型」的逻辑处理包；
2. 如果前端要求停止 VPN，则进入失败／关闭流程；
3. 如果前端提供了虚接口 fd，则设定 `epoll`。

具体实现均可以参见 `app/src/main/cpp/communication.cpp` 文件；与前端的通信部分参见 `app/src/main/cpp/frontend_ipc.cpp` 文件。

服务器

服务器的基本流程与客户端后台一致，除了以下区别：

后台整体流程

1. 在一开始需要利用 `/dev/net/tun` 创建虚接口；
2. 没有与「前端」的通信；
3. 需要 `listen` 接入，并对 `accept` 的 `socket` 进行主循环；

主循环逻辑等

1. 在接到 IP 请求时，应当返回分配的 IP 地址；
2. 处理的 over6 包类型不同；
3. 封装的 over6 包为 103 类型。

具体实现，可以参见 `DroidOver6Server.c` 文件。

前端实现

Android前端包括一个活动（Activity），用于呈现图形界面，以及一个服务（Service），该服务类继承自VpnService类，是在Android系统上实现VPN的必要步骤。服务启动之后，会启动一个额外的定时器线程。该定时器会每隔一秒向后端通过管道查询后台运行状态和流量统计信息，并通过Android的广播（Broadcast）机制将结果发送给主界面。主界面除了现实相关的流量统计信息、IP配置之外，还允许用户在连接之前配置远端服务器的IP地址和端口。

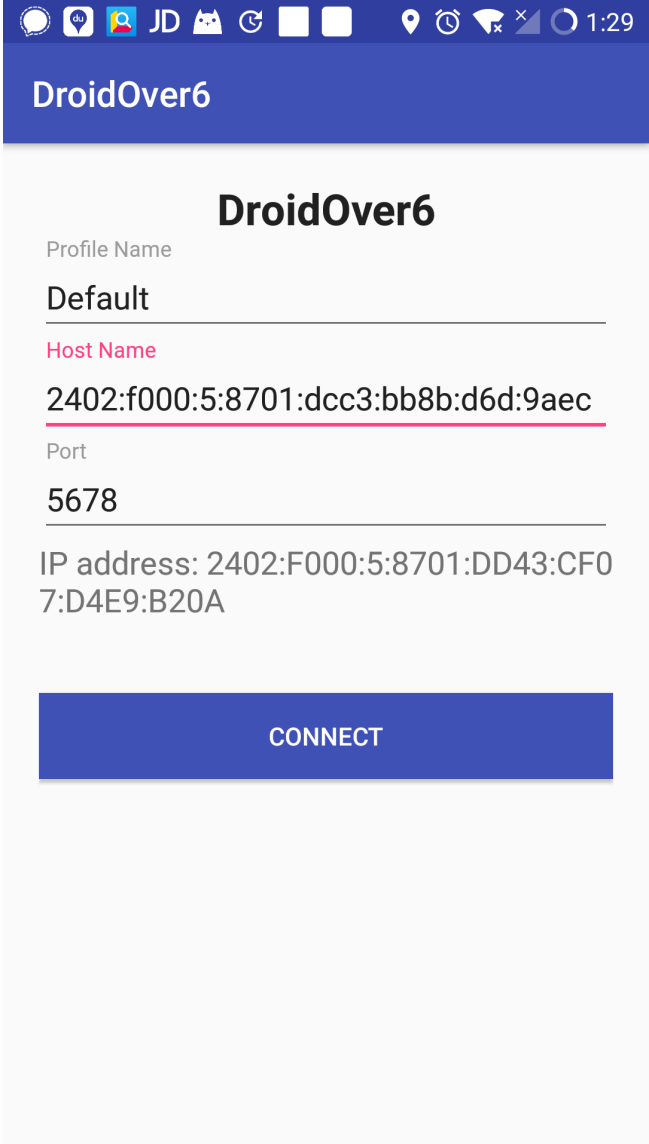
当由于服务器退出等原因造成后端检测到连接断开时，后端线程会退出。此时服务计时器会检测到这一问题，并且通知图形界面做出对应的更新。用户也可以直接通过图形界面上的按钮断开连接。

实验结果

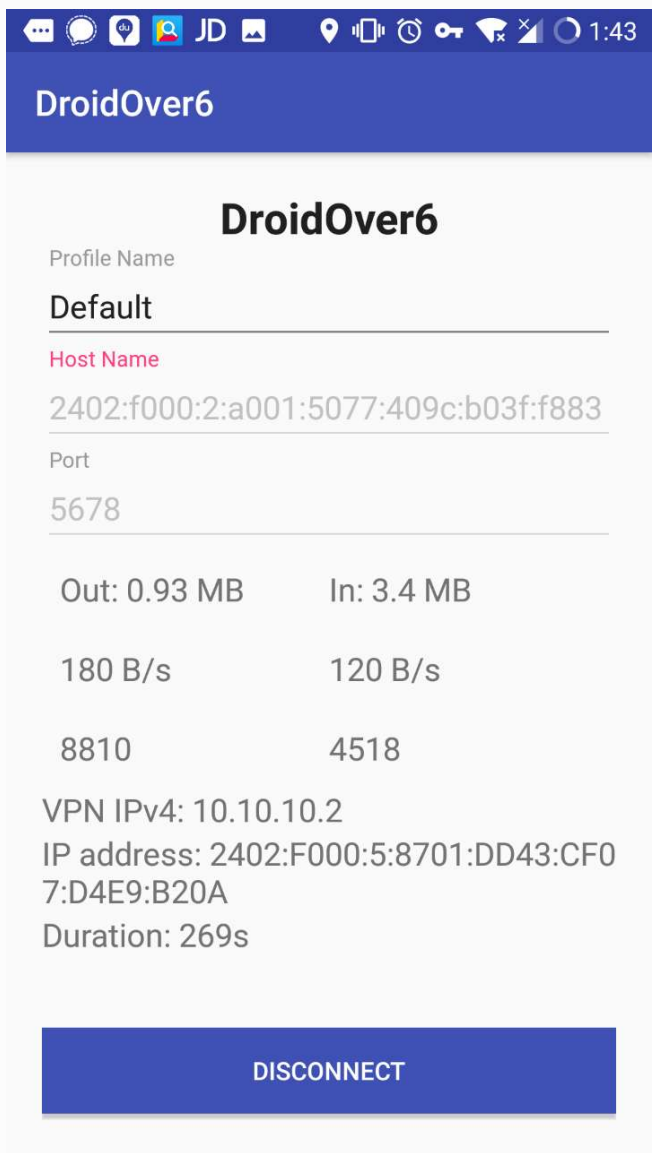
运行情况

前端界面

VPN服务启动前：



VPN服务运行中:



正常上网

新浪的花边新闻：

JD

11

1:43

https://sina.cn

今日要闻

高清美图

精选视频



自助餐厅搞人体宴 美女模特突然跳起来猛揍男

6张



蒋欣捞金后踩拖鞋回酒店 挽帅哥有说有笑

4张



莫斯科遭遇暴风雨 11人死亡50人受伤

4张




菲男子患罕见怪病 皮肤如鳞片、日渐变硬

4张



新生儿出生便会行走，近亿网友围观

4张



17张

通过Bilibili观看动画：



测试结果

网络测试结果

在我们自己的实验环境（清华宿舍楼 IPv6），网络测试情况如下：

1. 能够正常连接我们实现的简易 4over6 服务器，通信流畅；
2. 能通过网络中心的 especially 账号上网；
3. 能够正常浏览 Baidu.com；
4. 能够流畅观看 Baidu.com 以及 Bilibili.com 的视频；
5. 服务器端利用 `tcpdump` 查看 TCP 连接，以及虚设备通信数据，表现正常，数据包 checksum 全部正确。

在实验室现场（东主楼 DIVI Wi-Fi），网络测试情况如下：

1. 能够正常连接 4over6 服务器，包括我们实现的简易服务器以及课程提供的测试服务器，通信质量可以接受；
2. 连接我们的服务器，能够通过网络中心的 especially 账号上网；使用课程提供的测试服务器，可以连接 net 上网；
3. 能够正常浏览 Baidu.com；
4. 可以观看视频，但不流畅。

不流畅的原因我们分析为 DIVI Wi-Fi 的不稳定导致的，分析的基础为使用 DIVI 提供的 IPv4 也无法获得流畅的网络体验。

功能测试情况

在实验室现场，前台功能测试结果如下：

1. 能够正常启动后台服务，不阻塞前台显示；
2. 能够通过输入 IP 地址和端口，连接到正确的服务器，并且输入的信息能够记录到之后的使用；
3. 能够在点击连接时连接到 4over6 服务器，开启 VPN 服务，并在点击断开时结束连接，关闭 VPN 服务；
4. 在掐断服务器 4over6 服务的时候，前台能够显示连接断开的提示，关闭 VPN 服务；
5. 能够正常地与后端通信，发送任务、请求数据，提供创建的 VPN fd。
6. 能够正常地定期刷新界面上的显示，包括：
 1. 连接时长；
 2. 总包数；
 3. 总流量；
 4. 流量速度；
 5. 进行网络检测，显示本机 IPv6 地址；
 6. 4over6 获得的 IPv4 地址。

在实验室现场，后台功能测试结果如下：

1. 能正常与前端通过管道通信，获取虚接口 `fd`，读写 VPN 网络数据；
2. 能正常连接 4over6 服务器；
3. 能将前端所需的 4over6 网络 IP 信息、包数、总流量与速度信息和连接时长传予前端；
4. 能对数据进行 4over6 解封装，读写连接的网络数据；
5. 定时给服务器发送 keepalive 消息，并在服务器超时时关闭 VPN 连接。

在实验室现场，简易服务器功能测试结果如下：

1. 运行时能正确创建操作系统的 tun 虚接口，实现对虚接口的读写操作；
2. 能监听客户端接入，维护与隧道客户端的 IPv6 TCP 套接字；
3. 能读取客户端从 IPv6 TCP 套接字发送来的数据，实现对系统的控制消息和数据消息的处理；
4. 能对数据消息的解封装，并写入虚接口；
5. 能对虚接口接收到的数据报文进行封装，通过 IPv6 套接字发送给客户端；
6. 能监测客户端是否在线，及时断开，并且定时给客户端发送 keepalive 消息。

实验指导书的建议

TCP...?

TCP 并不是完全可靠的协议，因此长度段的传输可能出现错误，那么后面的包就无法正确分割，会对协议带来灾难性的后果。这是用单个不断开的 TCP 流传输「包」的天然缺点。解决方法只有再一次加入多余的纠错，增加 overhead。HTTP 即便使用了 TCP，早期也是一个请求来回一个 TCP 连接，即便是进入 keep-alive，也是与一个服务器的很少传输信息用一个连接，因此不会出现大的故障。

一般而言，实现 VPN 或者隧道都会优先考虑使用 UDP 或者直接作为三层协议，原因是 UDP 天然为包。并且其上传输的是三层，甚至于有时候可能是二层的数据，VPN 或者隧道并不需要保证数据的可靠性，即便是丢失或者损坏，二层以上，或者三层以上的容错机制也能使得链路正常运行。反倒是使用 TCP，其多余的纠错机制会增大传输的 overhead，提高成本，增大延时。

因此，希望未来这个实验改为使用 UDP 实现，更加符合现实世界的情况，降低了实现困难（如果要实现一个完备的 TCP 隧道，比 UDP 困难得多），也可以给学生带来工程上的思考。

4over6...?

虽然 4over6 有很多不一样的实现，但是这个.....说真的，实在不是一个正常的 4over6 实现。我建议在未来的课程中使用真实世界的 4over6 协议作为基础。作为一个原型验证，这个结构整体而言还是合格的，但是包的设计，尤其是 101 IP 回应包，还是值得再多加考虑。希望能在未来有所改进。

端序...

这是我第二次在这门课程中吐槽端序的问题了。在上次 IPv6 的转发实验中，我就看到了一些同学用了 32 位访问而没有考虑地址端序的问题。而这一次的项目中，我们的程序可以和自己的测试服务器正常通信，却在和实验提供的服务器通信上出了问题。

调查结果发现，`int length` 的传输中，对端的样例服务器没有转换为网络大端序，我们收到的数据为小端序数据。为了和这个写错的程序兼容，同时不违背初衷，我们规定 4over6 的数据为小端序，并且写了函数将长度转换为小端序。

使用网络序传输是互联网的一个重要的概念，我认为这门课程不应当直接忽视这一话题。希望在未来的课程中，能够要求同学们重视这个问题，注意这个问题。即便现在绝大多数机器为小端序，也不要忘记互联网互通互联的初衷。

int... char... packed...

来自 <http://en.cppreference.com/w/cpp/language/types> 的统计数据：

32 bit systems:

- **LP32** or **2/4/4** (int is 16-bit, long and pointer are 32-bit)
 - Win16 API
- **ILP32** or **4/4/4** (int, long, and pointer are 32-bit);
 - Win32 API
 - Unix and Unix-like systems (Linux, Mac OS X)

64 bit systems:

- **LLP64** or **4/4/8** (int and long are 32-bit, pointer is 64-bit)
 - Win64 API
- **LP64** or **4/8/8** (int is 32-bit, long and pointer are 64-bit)
 - Unix and Unix-like systems (Linux, Mac OS X)
- **ILP64** or **8/8/8**: int, long, and pointer are 64-bit)
 - early 64-bit Unix systems (e.g. Unicos on Cray)

int 不是定长的，只保证了大于 16 bits。char 也不是定长的，只保证了大于 8 bits。

请在以后的实验指导书中，将 int 修改为 uint32_t，将 char 修改为 uint8_t。

另外，如果结构体没有进行 packed 的话，在一些系统下会被 align，例如 gcc 下需要使用

`__attribute__((packed))`。请在以后的实验指导书中有所提醒。