4over6 实验报告

张翔　李家昊

2020 年 6 月 4 日

目录

1 客户端 **1**
1.1 实验原理 1
1.2 实验内容 2
1.2.1 后台流程 2
1.2.2 前台流程 3
1.2.3 前后台交互 3
1.3 实验结果 4
1.4 遇到的问题与解决方案 5
1.5 代码运行命令 5

2 服务端 & Linux 客户端 **5**
2.1 实验原理 5
2.2 实验内容 6
2.2.1 服务器工作流程 7
2.2.2 客户端工作流程 7
2.3 实验结果 7
2.4 遇到的问题与解决方案 8
2.5 代码运行命令 9

3 实验分工 **9**

1 客户端

1.1 实验原理

IPv4 over IPv6，简称 4over6，是 IPv4 向 IPv6 发展进程中，向纯 IPv6 主干网过渡的一种新技术，可以最大程度的继承基于 IPv4 的网络和应用，实现 IPv4 向 IPv6 的平滑过渡。

本实验实现了 4over6 隧道的最小原型验证系统，原理如图 1。4over6 隧道将 IPv4 报文封装在 IPv6 报文的数据段，经过过渡网关的封装和解封装，使 IPv6 用户访问到 IPv4 网络的资源。

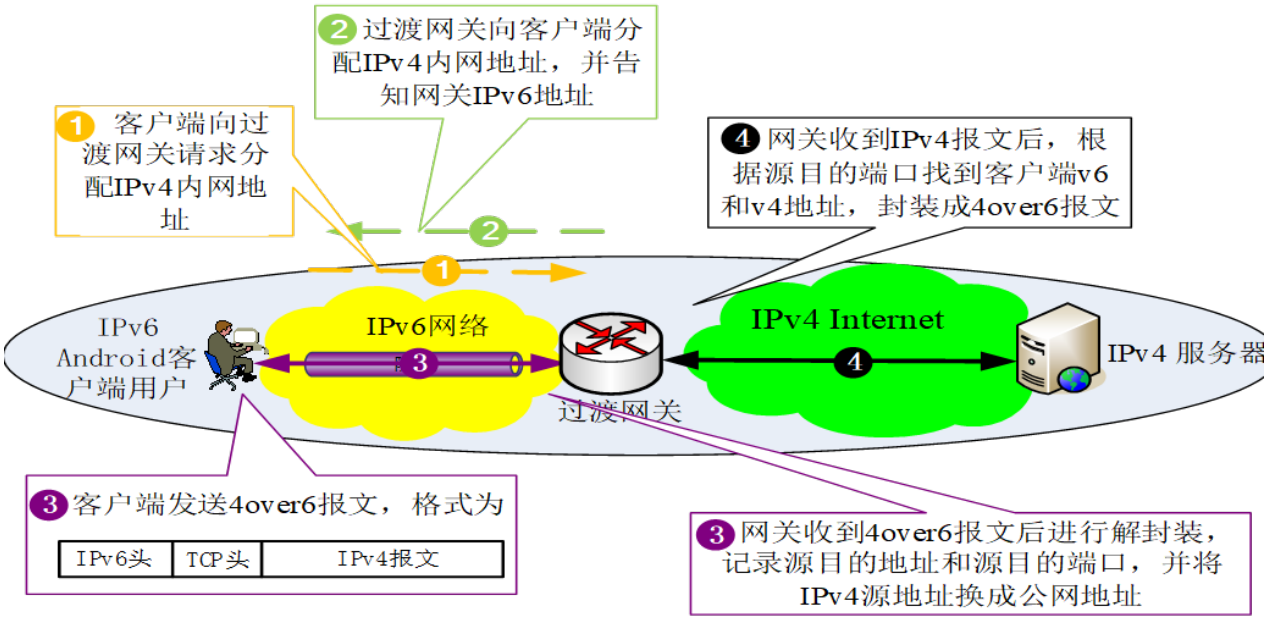


图 1: 实验原理

1.2 实验内容

客户端主要分为前台和后台两个模块。前台用 Java 语言实现，负责维护用户的 VPN 配置和显示界面；后台用 C++ 语言实现，主要实现了与服务端交互的相关接口，封装成动态链接库给前台使用。

1.2.1 后台流程

正常运行时，后台共有三个线程。

- 计时线程：负责计时并发送心跳包。当距离上次发送心跳包的时间超过 20 秒时，向服务端发送心跳包，并更新上次发送时间；如果距离上次收到心跳包的时间超过 60 秒，则断开与服务器的连接；上述过程每间隔 1 秒执行一次。
- 接收线程：负责接收并处理数据包。不断读取服务端发来的数据包，当类型为 101（IP 地址回应）时，则将 IP、路由、DNS 信息记录到全局变量，并通知正在等待的进程，当类型为 103（上网回应）时，则将数据段提取出来，作为 IPv4 报文写入 tunnel 虚拟设备中（`/dev/tun`），当类型为 104（心跳包）时，则更新上次收到心跳包的时间。
- 发送线程：负责发送数据包。不断从 tunnel 中读取 IPv4 报文，创建类型为 102（上网请求）的 4over6 报文，将 IPv4 报文作为数据段，转发到 socket。

同时，提供了以下接口给前台使用。

- `connect_socket`：根据指定的 IPv6 地址和端口，创建 IPv6 socket 并连接，若连接成功，则开启接收线程和心跳包线程，并返回 socket 描述符。
- `request_ipv4_config`：向 socket 写入类型为 100（IP 地址请求）的报文，并阻塞等待服务端回复。
- `setup_tunnel`：开启发送线程。

- `disconnect_socket`：终止三个线程，关闭 socket 和 tunnel，断开连接。
- `is_running`：判断后台是否正在运行。
- `get_ipv4_config`：获取后台记录的 IPv4 配置，包括 IP，路由和 DNS。
- `get_statistics`：获取后台的流量统计信息。

如果选择了加密，则采用 `libsodium` 的 chacha20-poly1305 加密算法进行对称加密，用 UUID 的 16 字节以及 `nonce` 的前 16 个字节组成 32 字节的密钥，实际传输的数据格式如下，其中 `encrypted 4over6 message` 段存储了加密后的原始 4over6 报文。

```
+--------+------------------+------------ data -----------------------+
| length | type = ENCRYPTED | nonce (24 B) | encrypted 4over6 message |
+--------+------------------+----------------------------------------+
```

1.2.2 前台流程

初次使用时，用户必须创建一个或多个 VPN 配置，每个配置指定了 VPN 别名，服务端的 IPv6 地址或域名，服务端端口，是否加密，加密的 UUID。创建完成后，配置将会通过 `SharedPreference` 写入持久存储中，下次打开应用时会自动加载，无需再次创建。

用户可以选择一个 VPN 配置，按下圆形按钮进行连接，此时，

- 前台将会抛出对话框询问用户是否建立连接，得到用户同意后，系统的其他 VPN 连接会被断开。
- 调用后台的`connect_socket`连接 4over6 服务端，开启后台的接收线程和计时线程，连接成功后获得 socket 描述符，该 socket 需要直接连接到物理设备上而不是 tunnel 虚拟设备，即被 `VpnService` 保护。
- 调用`request_ipv4_config`阻塞获取 IPv4 配置，作为 `VpnService` 的配置参数，并开启 `VpnService`，得到 tunnel 描述符。
- 调用`request_ipv4_config`将 tunnel 描述符传到后台，开启后台的发送线程。至此，4over6 隧道已经建立。
- 前台开启计时器，每隔 1 秒获取后台的上行和下行流量统计信息，计算出实时带宽并更新到界面上。

用户断开连接时，调用`disconnect_socket`断开后台连接，关闭 `VpnService`，并更新界面。

用户暂时离开，然后回到应用时，前台页面可能已经被销毁，此时需要查询后台的运行状态，将前台恢复到相应的运行状态。

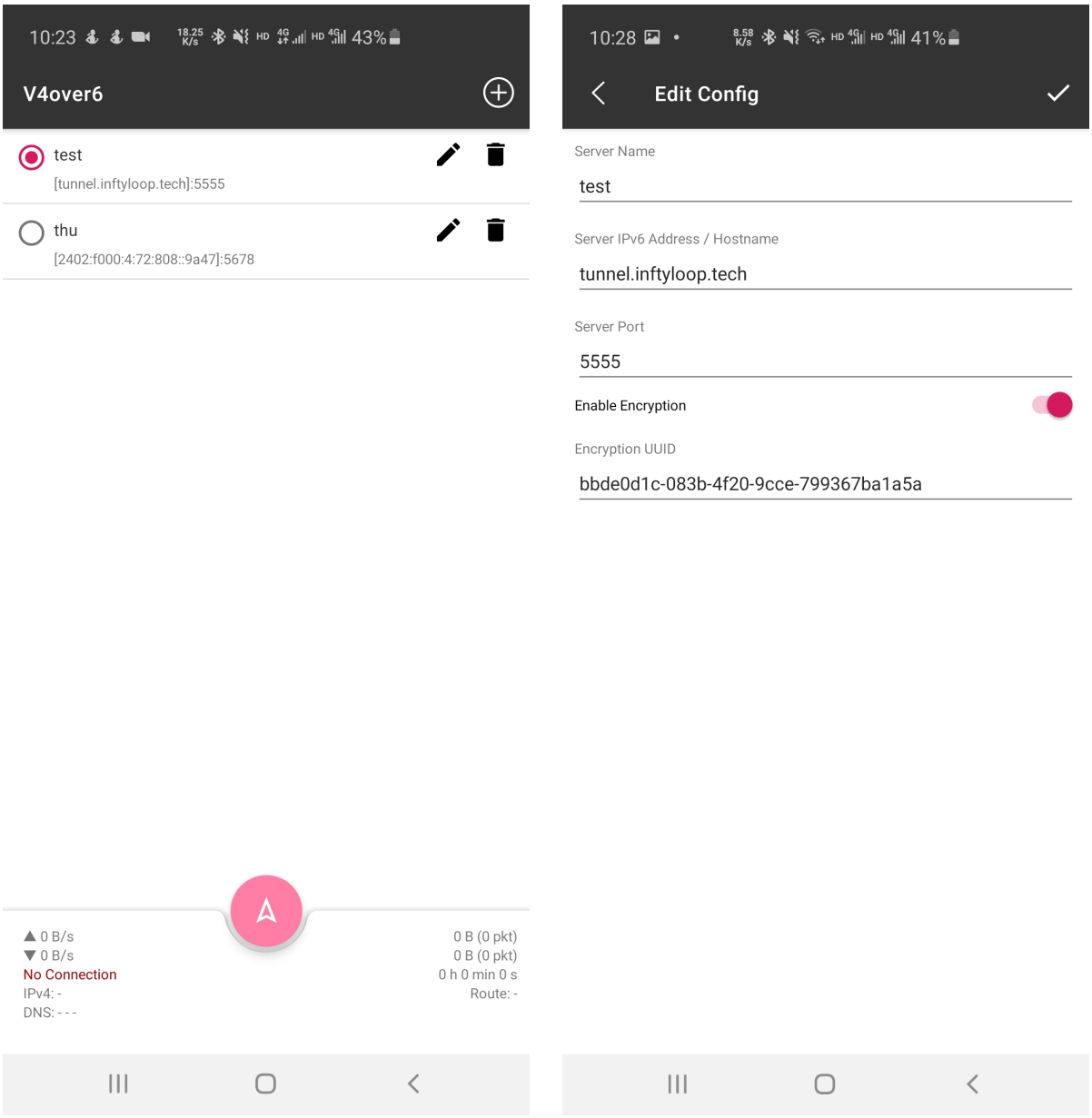
需要注意的是，所有的阻塞操作都必须在子线程中进行，避免主线程（UI 线程）阻塞。

1.2.3 前后台交互

编译时将后台封装成动态库，提供相应接口给前台使用。运行时前台通过 JNI 调用后台接口，与服务端交互，并获取后台的统计数据。需要注意的是，前后台运行在同一个进程之中，因此两者可以直接交换数据，效率较高，不必通过管道等跨进程通信机制进行通信。

1.3 实验结果

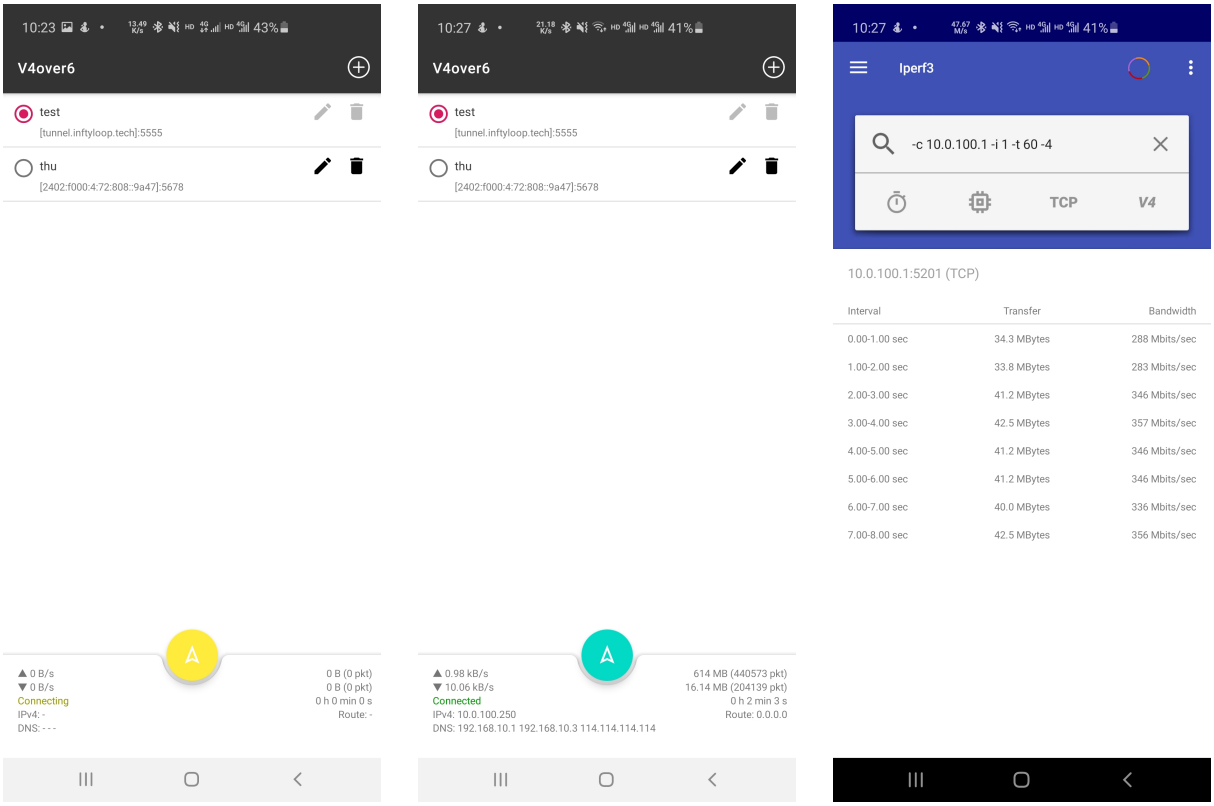
进入应用时，主界面为 VPN 配置列表，如图 2(a)，每个配置摘要显示了 VPN 别名、服务端地址和端口。用户可以新增或修改 VPN 配置，界面如图 2(b)，可以设置 VPN 别名，服务端 IPv6 地址或域名、端口、是否加密、加密的 UUID。



(a) 主界面　　(b) VPN 配置界面

图 2: 主界面和 VPN 配置界面

配置完成后，用户可选择一个 VPN 配置进行连接，如图 3(a)和图 3(b)。连接成功后，前台显示下联虚接口的 IPv4 地址，并持续更新流量信息。使用 Network Tools 安卓 APP 的 `iperf3` 工具，测试客户端到下联虚接口网关的带宽，结果如图 3(c)。



(a) 连接中　　(b) 连接成功　　(c) `iperf3` 测速结果

图 3: VPN 连接过程及测速结果

1.4 遇到的问题与解决方案

1. 开启 `VpnService` 后，不能向外发送任何数据。原因是系统的所有流量都通过 iptables 转发到 tunnel 虚拟设备上，包括 socket 上的流量，所以数据永远在本地回环而无法流出。解决方法是使用 `VpnService.protect` 来保护 socket，使 socket 直接连接物理网络设备，而不经过 VPN。

2. 在建立连接时用户界面卡顿，甚至无响应，导致被系统强制关闭，这通常是在访问其他开放的 IPv6 服务时发生的。原因是建立连接后，会一直阻塞等待服务端发送 IPv4 配置，如果服务端在运行其他网络协议而非 4over6 时，就一直不会发送，直到心跳包超时断开连接。解决方法是将连接过程放在子线程中执行，连接时界面处于 `CONNECTING` 状态，保证 UI 线程不卡顿，同时在等待过程设置 Timeout，超时则返回失败。

1.5 代码运行命令

使用 Android Studio 编译运行即可。

2 服务端 & Linux 客户端

2.1 实验原理

4over6 的原理请参考客户端1.1部分。对于服务端，它的架构如下图

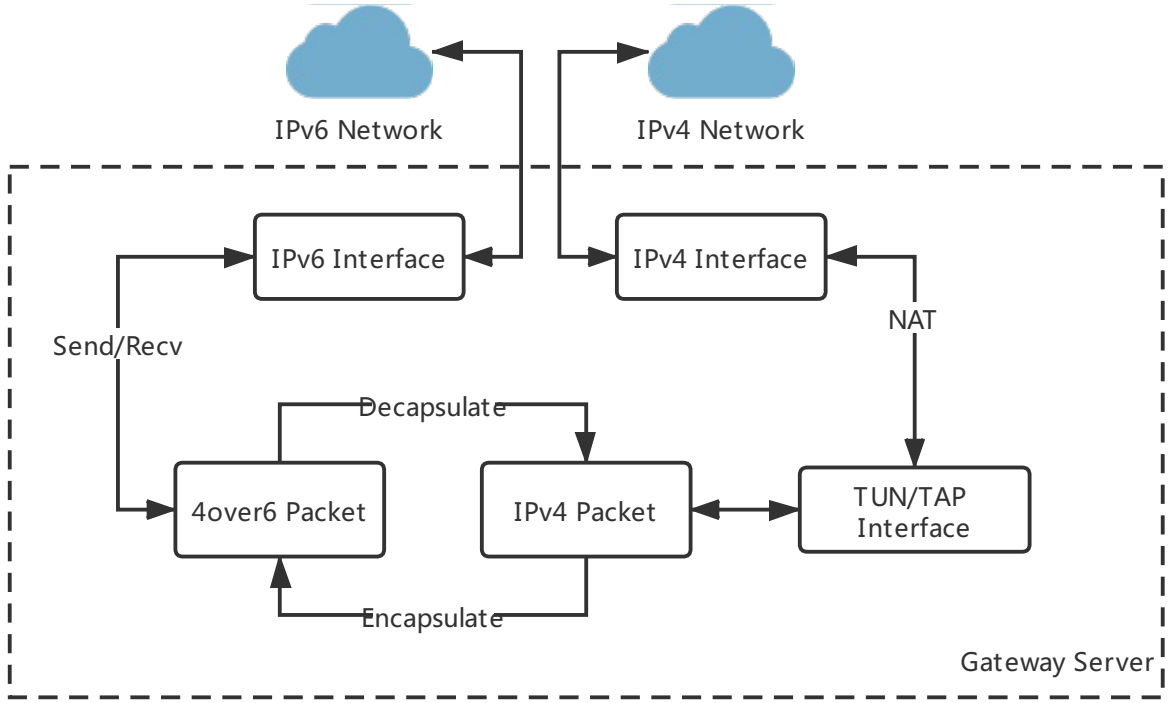


图 4: 服务端架构图

客户端通过 IPv6 网络接入网关服务器，服务器建立 TCP Socket，监听客户端的连接请求。4over6 协议数据包从 IPv6 网络流入，经网关服务器解包为 IPv4 报文，发送至服务器建立的虚接口。之后该报文由 Linux 的网络栈处理，通过地址转换 (NAT) 后从 IPv4 默认路由指定的接口传输至外部 IPv4 网络。当 IPv4 的接口出收到数据后，如果在 NAT 表中存在相应映射，就会将报文发送至虚接口，网关程序读取后，将其封装为 4over6 报文，通过 IPv6 网络传送至客户端。

注意这里的 NAT 不是必需的，如果客户端不需要访问 Internet，而是通过 4over6 服务器接入某个 IPv4 局域网中，只需要开启 Linux 的转发功能，设置好路由表即可。当报文到达虚接口后，之后的工作流程和普通路由器的物理接口是类似的。

2.2 实验内容

这里使用了 C++ 的 Boost 库作为基础开发了服务端和 Linux 客户端，采用的是异步编程模型（使用 Boost ASIO），不需要开启多个线程。诸如 Socket 读取时数据就绪的情形会生成一个事件，放入事件队列中，由事件循环加以处理。

服务器与 Linux 客户端共享了以下的类

- `TunDevice`: 封装了 Linux 的虚接口，支持分配 IP、MTU 等参数，并设置路由，提供异步读接口 `async_read_packet` 与同步写接口 `send_packet`
- `SecurityHandler`: 加/解密工具类，初始化时传入 16 Byte 的 UUID，它会调用 `libsodium` 的 chacha20-poly1305 加密算法对 4over6 的 Message 进行加/解密

服务器端设计了如下类

- `AddressPool`: IPv4 地址池，初始化时传入 IPv4 地址范围，服务器需要时可以从中请求 IP (`obtain_ip_address`)，或释放 IP(`return_ip_address`)

- `ServerSession`: 维护服务器与某个客户端的连接信息，并处理客户端发送的报文，或将虚接口的报文封装后发给客户端
- `Server`: 服务器本体类，维护全局用户信息，处理传入连接并分配 IP、创建 Session

对于 Linux 客户端，它只有 `Client` 类，用于处理与服务器的连接。

由于采用异步编程，下面使用事件-响应的方式来描述工作过程

2.2.1 服务器工作流程

- 传入连接 –> 创建与客户端通信的 Socket，尝试从地址池获取 IP，若获取成功，创建 Session，将用户的 IPv6 与分配的 IPv4 绑定，开始异步处理；若获取失败，关闭连接
- 虚接口数据就绪 –> 查询用户的 IPv4 与 Session 对应关系，若存在，调用 Session 的 `write` 方法将报文封装为 4over6 消息，并加密（若启用加密）发送到客户端
- 心跳包检查超时（由一个 1s 的计时器触发）–> 遍历用户表，如果遇到心跳包接收超时的用户，关闭 Session（相应 Socket 会自动被关闭），释放 IPv4 地址，并清除表中的对应关系；否则将用户的 count 字段减 1，当减到 0 后向用户发送心跳
- 用户 Socket 数据就绪 –> 对数据包进行解密（若数据是加密的），根据消息类型进行分发：如果是 IP 请求，返回创建 Session 时分配的 IPv4 地址与配置的 DNS；如果是心跳包，修改上次收到客户端心跳包的时间；如果是上网请求，将它解包为 IPv4 报文发送给虚接口

2.2.2 客户端工作流程

客户端的代码从服务端代码修改得到，它的工作流程如下

- 与服务器连接的 Socket 收到数据 –> 对数据包进行解密，根据消息类型分发：如果是 IP 响应，就根据信息对虚接口的 IP 进行配置；如果是心跳包，修改上次收到服务器心跳的时间；如果是上网响应，将它解包后送给虚接口
- 虚接口数据就绪 –> 封装成 4over6 消息，加密后发送给服务器
- 1s 计时器超时 –> 如果距离上次发送心跳超过 20s，向服务器发送心跳；如果距离上次收到心跳超过 60s，断开与服务器的连接。操作完成后重新设置计时器，保证 1s 后能够异步回调心跳检查函数

2.3 实验结果

下图是服务器开启并接受了 1 个客户端的示意图

```
[root@seafile-server build]# ./server -c ../server_config.yaml
2020-06-04 22-37-11.152 [Info] (main) Starting server with encryption support
2020-06-04 22-37-11.152 [Info] (assign_tun_route) Add route: 10.0.100.1, mask: 255.255.255.0
2020-06-04 22-37-15.170 [Info] (handle_client) IPv4 lease: 10.0.100.250, remaining: 248
2020-06-04 22-37-15.170 [Info] (start) Accepted client: 240e:360:6f10:d500:a871:e45e:88aa:1b6b:37000
```

图 5: 服务端开启时的界面

服务端开启时会自动创建一个虚接口并为它配置 IP 与 MTU，如图

```
tun0: flags=4565<UP,DEBUG,POINTOPOINT,RUNNING,NOARP,PROMISC,MULTICAST>  mtu 1500
        inet 10.0.100.1  netmask 255.255.255.255  destination 10.0.100.1
        inet6 fe80::da80:7a04:182f:60b4  prefixlen 64  scopeid 0x20<link>
        unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00  txqueuelen 500  (UNSPEC)
        RX packets 443528  bytes 611968881 (583.6 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 207385  bytes 16503506 (15.7 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 6: 服务器创建的虚接口

Linux 客户端连接服务器如图所示

```
zx@zx-virtual-machine ~/4over6/build master ⊙1 sudo ./client -c client_config.yaml SIGINT(2) ↵ 2584 22:37:08
2020-06-04 22-37-15.149 [Info] (main) No pre-shared key in config file, disabling encryption support
2020-06-04 22-37-15.149 [Info] (start) Connecting to server: 240e:360:6f10:d500:707a:8920:7852:867a
2020-06-04 22-37-15.149 [Info] (start) Successfully connected to server
2020-06-04 22-37-16.149 [Info] (get_ip_config) Sending IP request to server
2020-06-04 22-37-16.149 [Info] (on_data_read_done) Got config: [IP] 10.0.100.250, [ROUTE] 0.0.0.0, [DNS] 192.168.10.1,192.168.10.3,114.114.114.114
2020-06-04 22-37-16.151 [Info] (assign_tun_route) Add route: 10.0.100.250, mask: 255.255.255.0
```

图 7: Linux 客户端连接的界面

在网关服务器上开启 `iperf`，在客户端测试传输性能，结果如下

```
zx@zx-virtual-machine ~/4over6/build master ⊙1 iperf3 -c 10.0.100.1
Connecting to host 10.0.100.1, port 5201
[  5] local 10.0.100.250 port 53348 connected to 10.0.100.1 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[  5]   0.00-1.00   sec  88.2 MBytes   740 Mbits/sec  737   1.13 MBytes
[  5]   1.00-2.00   sec   112 MBytes   944 Mbits/sec    0   1.20 MBytes
[  5]   2.00-3.00   sec   115 MBytes   964 Mbits/sec  246    703 KBytes
[  5]   3.00-4.00   sec   122 MBytes  1.03 Gbits/sec    0    827 KBytes
[  5]   4.00-5.00   sec   118 MBytes   986 Mbits/sec  111    676 KBytes
[  5]   5.00-6.00   sec   119 MBytes   996 Mbits/sec    0    802 KBytes
[  5]   6.00-7.00   sec   118 MBytes   986 Mbits/sec   95    642 KBytes
[  5]   7.00-8.00   sec   130 MBytes  1.09 Gbits/sec    0    785 KBytes
[  5]   8.00-9.00   sec   128 MBytes  1.07 Gbits/sec    0    901 KBytes
[  5]   9.00-10.00  sec   120 MBytes  1.01 Gbits/sec    0    997 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[  5]   0.00-10.00  sec  1.14 GBytes   981 Mbits/sec  1189             sender
[  5]   0.00-10.01  sec  1.14 GBytes   978 Mbits/sec                  receiver

iperf Done.
```

图 8: `iperf` 性能测试 (Linux 客户端)

从图中可以看出，性能基本上可以跑满千兆带宽。

2.4 遇到的问题与解决方案

1. 客户端发送的数据包服务器均能正常接收，但服务器向客户端大量发送时，客户端收到的却是乱序数据（或客户端可以 `ping` 通网络，但大流量下载时传输失败）：使用 `Wireshark` 抓包，可以看到客户端的 IPv4 协议栈发送了大量的 TCP Retransmission 报文，说明出现了服务器将数据回传给

客户端的代码有一定问题。检查发现服务器代码中出现连续调用 `async_write` 的情况，而 Boost ASIO 要求异步写调用必须序列化，即前一个异步写操作未回调前，不能再执行一个异步写操作（`write`），否则数据包可能会乱序。解决方法：采用同步写操作或手动维护一个写队列，为了实现方便，采用了前者。

2. 服务端收到一些无效数据包，它的类型 ID 不在协议规定的范围内，消息长度也不对。解决方法：Boost ASIO 的 `async_read_some` 只要有部分数据就绪时就会回调，考虑到 TCP 具有 stream 的特性，使用 `async_read` 指定每次读取的大小（该函数读取完指定大小的数据后才会回调），先读取消息头，再根据消息头中指定的大小来读取消息体即可。

2.5 代码运行命令

请参考 `server` 目录下的 `README.md` 文件。安装完依赖后，使用 `mkdir build && cd build && cmake .. && make` 进行编译，编译完成后即可运行。如果需要允许客户端访问公网，可以执行 `server` 目录下的 `nat.sh`，它会自动检测公网接口，并设置 IPv4 转发与 NAT。

3 实验分工

- 张翔负责 Linux 服务端与 Linux 客户端部分
- 李家昊负责 Android 客户端部分