4over6 服务器实验报告

实验人：陈熠豪

学号：2017011486

邮箱：chenyiha17@mails.tsinghua.edu.cn

实验原理

- 主程序 src/main.c [main]
    - 注册信号处理函数
    - 初始化服务器
    - 初始化隧道
    - 创建和注册 epoll 事件
    - 初始化用户信息表
    - 进入事件处理循环
        - 监听 socket 响应
        - 隧道 socket 响应
        - 用户 socket 响应
        - 心跳检查
- 错误处理 src/main.c [error_handler]
    - 注册 SIGABRT 信号处理函数，打印调用栈
- 初始化服务器 src/Server.c [init_server]
    - 建立 AF_INET6 的套接字并设置地址重用
    - 绑定 ip 地址和端口，并进行监听
- 初始化隧道 src/Tun.c [init_tun]
    - 打开隧道设备，命名和设置虚拟网卡
    - 调用系统命令，绑定虚拟网卡的 ip 地址，并建立 nat 规则

      (如果已经有现成的隧道，也可以设置宏 SET_TUN=0 跳过这一段；可参考运行 tun_setup.sh 建立永久隧道)
- 创建和注册 epoll 事件 src/Epoll_Utils.c [epoll_init, epoll_add_fd]
    - linux 内核提供了 epoll 服务，可用于检测高并发场景下的套接字事件
    - 调用库函数建立 epoll 句柄，并将创建的服务器监听 socket、隧道 socket 加入 epoll 控制队列

- 初始化用户信息表 src/User_Info.c [user_info_init]
    - 初始化用户信息的数组，分配对应的 ip
    - 初始化用户对应的 buffer
- 监听 socket 响应 src/Server.c [process_server]
    - 接收新的 socket，从而和用户建立连接
    - 将用户 socket 加入 epoll 控制队列
- 隧道 socket 响应 src/Tun.c [process_tun]
    - 从隧道 socket 中读取数据到对应的 buffer 中
    - 从 buffer 中取出一个 ip 包，检查目的 ip 地址是否在用户信息池中，如果不在，则将这个包丢掉；否则，根据 4over6 的通讯协议为这个包加上 Msg 头后，传送给对应用户的 socket
- 用户 socket 响应 src/User_Info.c [process_client]
    - 从用户 socket 中读取数据到对应的 buffer 中。如果读取长度为-1，则说明用户断开连接或者发生了错误，进行错误处理，清除对应用户的信息，回收地址
    - 从 buffer 中取出 Msg 头，根据 Msg 类型分别处理
        - 客户端 IP 地址请求（100）。为其分配用户信息条目，初始化 buffer，并将对应的 v4 地址以及 DNS 信息等通过 101 类型 Msg 传回
        - 上网请求（102）。将包的数据部分写入隧道
        - 心跳包（104）。搜索该用户条目，更新其上次心跳的时间
        - 其他非法的包直接丢弃
- 心跳检查 src/Keep_Alive.c [process_heartbeat]
    - 遍历所有用户条目，如果该条目有效，检查距上次发送心跳的时间是否超过了间隔，如果是，则向其发送心跳包；检查距上次收到心跳的时间是否超过了间隔，如果是，则将该条目清除，将对应 socket 关闭并移出 epoll 控制队列

实验内容

- 建立服务端运行环境
    - 运行 tun_setup.sh：建立隧道，设置相应的 nat 规则和内核转发
- 编译并运行服务端程序
    - 运行 run.sh：编译并运行
- 由客户端接入，实现 4over6 包转发服务

实验结果

- 服务端可以正确实现所有需求。性能结果请见客户端报告（可能记录了上下行带宽，也可能没有）

实验中遇到的问题

- 编程实现上出现的问题
    - 通过 tun 读取数据的问题

      我一开始在实现的时候，对于一个 ip 包分开两次读取，即第一次读取 ip Header，第二次再根据 ip Header 中的长度读取剩余的数据。这种方式经常（但不是每次）会出现第二次读取阻塞的情况，即第二次读取时缓冲区中已经没有数据了

      事实上，完全是没有必要分成两次读取的，系统内核的网络协议栈已经将数据包处理好了，直接调用 read 就能够读出完整的数据，检查一下读出数据长度和 ip header 中的长度是否一致即可

      至于为什么分开两次读会出现阻塞，可能是因为第一次读后会因为内核的某些抢占机制（考虑到这个问题不是稳定复现的）导致缓冲区被清空，所以第二次就读不出数据了

- 服务器环境配置和部署中出现的问题
    - 如何确定 DNS

      根据实验指导书，服务器 DNS 地址是通过读取/etc/resolv.conf 文件中的 nameserver 地址来获取的，但在实际的部署中发现，服务器上/etc/resolv.conf 中 nameserver 字段为 127.0.0.53，是一个本地地址，而/etc/resolv.conf 本身只是 systemd-resolve 的 symlink，并没有显示具体使用的 DNS。

      如果需要显示使用的 DNS，可以通过 systemd-resolve --status 来查看。当然，在实际的部署中，直接使用公共 DNS 就可以了，本实验最后使用了清华 DNS

    - 如何建立隧道

      我起初通过 tun_setup.sh 来建立永久隧道和设定 nat 规则，但是这样建立的隧道不够灵活，需要手动设置和比对 c 程序和脚本中的参数，也容易造成重复配置的问题；参考了一些开源代码后，我选择将设立隧道的过程放在 c 程序里进行，这样只要运行 c 程序就可，不需要每次提前设定和检查隧道，而且通过 c 程序建立的隧道的生命周期和 c 程序是一致的，也不会造成重复配置的问题

      但是在服务器机器重启后，我发现通过同样的方法建立的隧道没法使用了，往里面发送包后收不到任何返回的包。经过漫长的检查后，我发现是由于服务器重启，内核转发的配置被重置，导致虚拟网卡转发失效，而在运行 sudo sysctl -w net.ipv4.ip_forward=1 后问题就解决了

    - 如何进行网络 debug

      在整个实验过程中，出现问题后如何进行 debug 也是一个问题。比如在服务器重启后，隧道突然没有传回数据了，一方面需要对自己写的代码进行 debug，另一方面更需要对网络环境和配置进行检查。前者较为简单，我主要通过 debug 打印就能判断程序逻辑是否正确，而后者则较为麻烦，我的方法是使用 tcpdump 抓取特定网卡和网段的包，传回本地后再使用 wireshark 进行分析。我正是通过这种方法发现了虚拟网卡没有收到任何从物理网卡发回的包，进而猜测内核转发失效，从而发现了问题的具体原因

实验分工

- 客户端：李嘉尧
- 服务端：陈熠豪

代码运行命令

- 请见 Readme.md
