网络传输机制实验二报告

李昊宸

2017K8009929044

（一）TCP 稳定传输实现

一、实验内容

1. TCP server client 实验：

   1）运行网络拓扑(tcp_topo.py)

   2）在节点 h1 上执行 TCP 程序

   执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能

   在 h1 上运行 TCP 协议栈的服务器模式

   3）在节点 h2 上执行 TCP 程序

   执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能

   在 h2 上运行 TCP 协议栈的客户端模式，连接 h1 并正确收发数据：client 向 server 发送数据，server 将数据 echo 给 client

2. TCP server client 文件传输实验：

   1）运行网络拓扑(tcp_topo.py)

   2）在节点 h1 上执行 TCP 程序

   执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能

   在 h1 上运行 TCP 协议栈的服务器模式

   3）在节点 h2 上执行 TCP 程序

   执行脚本(disable_tcp_rst.sh, disable_offloading.sh)，禁止协议栈的相应功能

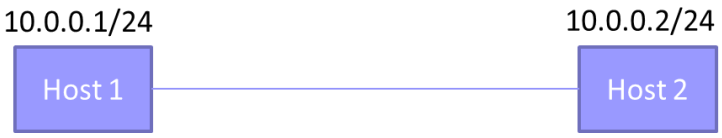
   在 h2 上运行 TCP 协议栈的客户端模式：Client 发送文件 client-input.dat 给 server，server 将收到的数据存储到文件 server-output.dat

4）比较两个文件是否完全相同

二、实验流程

1. 搭建实验环境

```
arp.c  arpcache.c  icmp.c  ip.c  main.c  packet.c  rtable.c
rtable_internal.c
tcp_apps.c  # 能够进行收发数据的 tcp sock apps
tcp.c           ：  TCP 协议相关处理函数
tcp_in.c        ：  TCP 接收相关函数
tcp_out.c       ：  TCP 发送相关函数
tcp_sock.c      ：  tcp_sock 操作相关函数
tcp_stack.py    ：  python 应用实现，用于测试
tcp_timer.c     ：  TCP 定时器
create_randfile.sh  # 随机生成文件的脚本
tcp_topo.py     ：  实现二节点的简单拓扑
```



图一 二节点网络拓扑

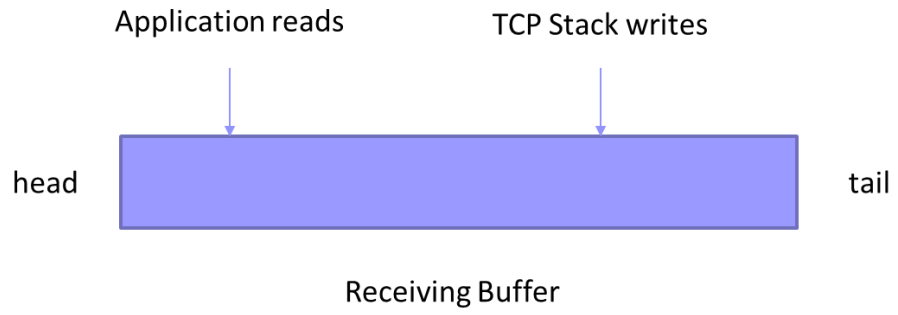
上周的实验已经实现 TCP 连接的建立和关闭，这周需要补充在连接建立后进行数据的传输。

TCP 中数据接收和缓存：TCP 协议栈收到数据包后，使用接收缓存来存储相应数据，供应用程序读取。

1）使用环形缓存(ring buffer)来实现

2）接收缓存大小为 recv_window

3）使用锁（pthread_mutex_t）来防止读写冲突



TCP 数据发送流程：

1）待发送数据全部存储于上层应用 buffer 中

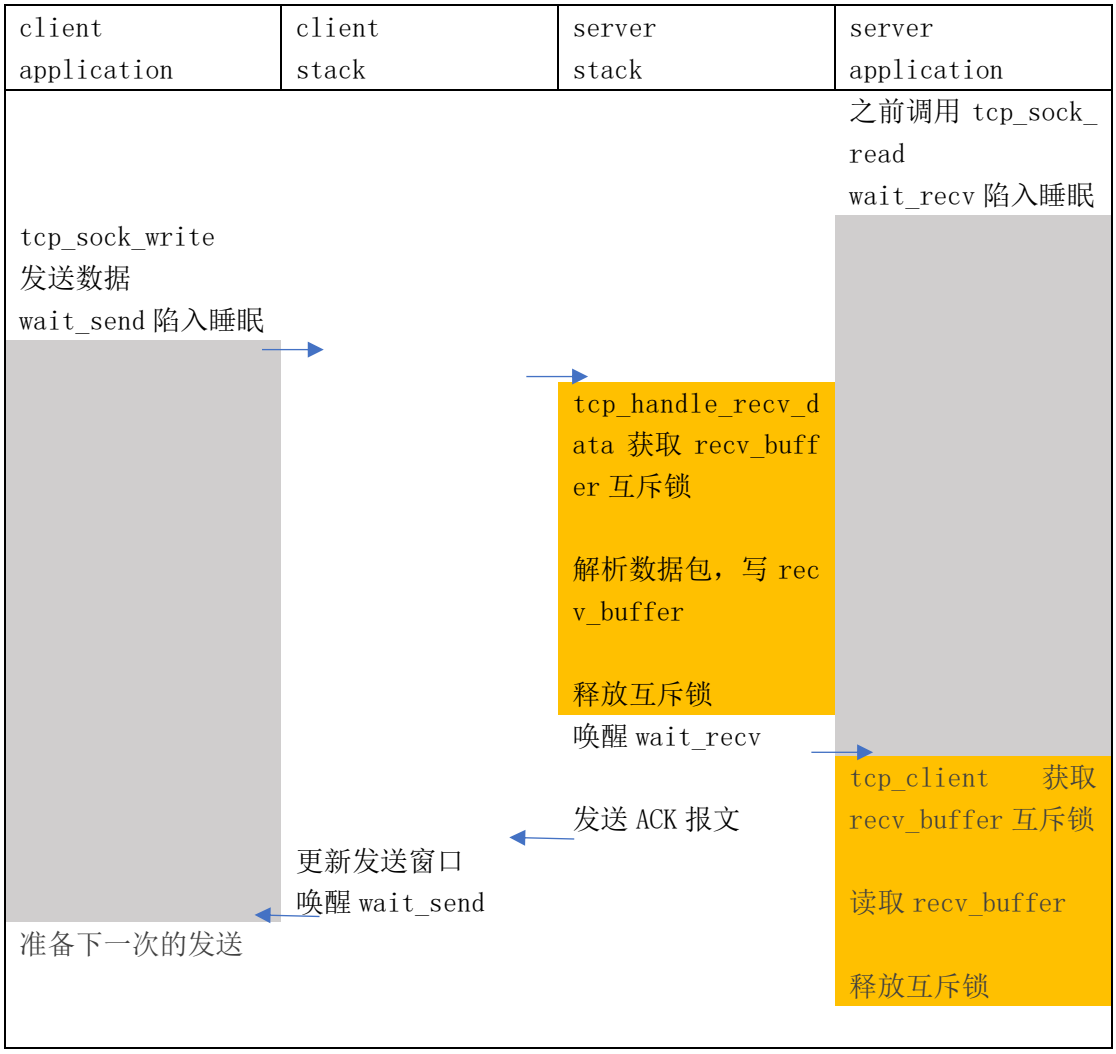
2）如果对端 recv_window 允许，则发送数据

3）每次从 buffer 中读取 1 个数据包大小的数据，封装数据包，通过 IP 层发送函数，

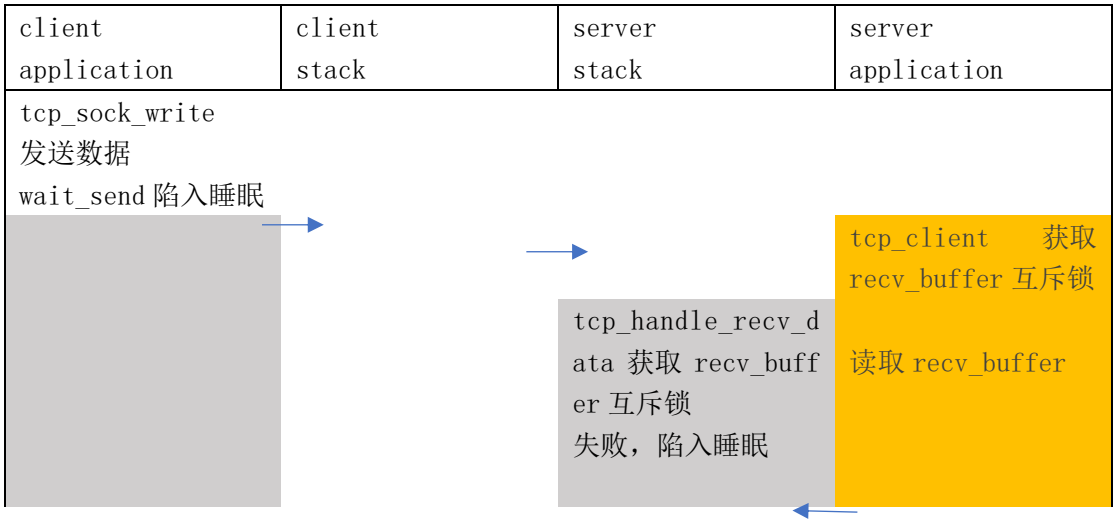
将数据包发出去

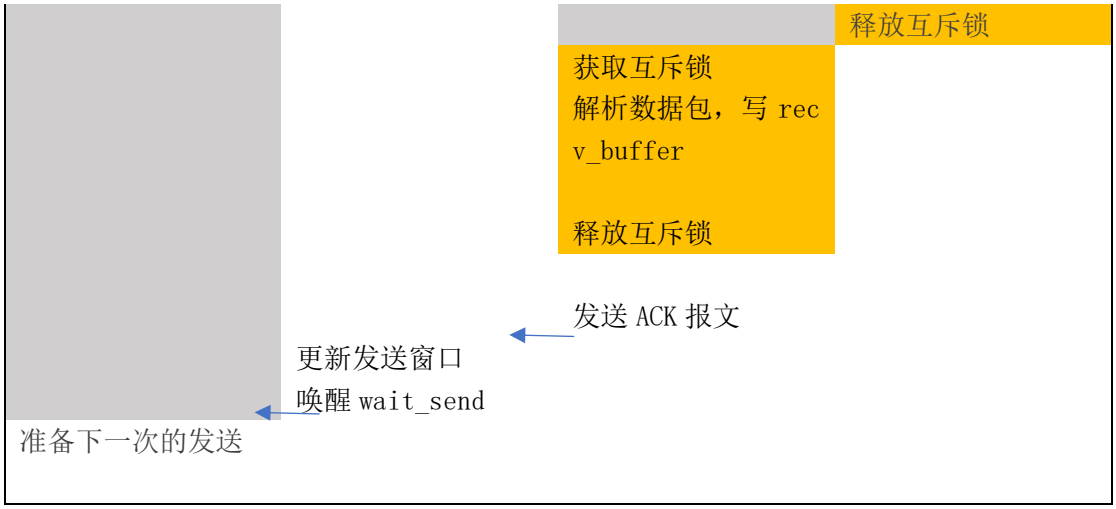
本实验中 TCP 两端时序图：

1）如果发送时，接收方（server）已经准备接收



2）如果发送时，接收方（server）还在处理之前的接收信息





本次实验的交互流程：

1） server 端首先新建 tcp_socket，绑定指定本地地址和端口，对其监听，调用 accept 操作等待请求到达

2） client 端首先新建 tcp_socket，调用 connect 操作请求 server 服务

3） server 监听到请求，如同上周实验所做的，新建子 tcp_socket，与 client 请求进行交互，最后建立连接

4） client 打开要发送的文件，将文件内容读取到缓冲区 buffer 中；server 打开要保存的文件，调用 tcp_sock_read 操作等待要到达的数据（wait_recv）

5） 重复以下操作：

client 调用 tcp_sock_write，将 buffer 中的部分数据发送出去，然后陷入等待（wait_send）。

server 接收到数据包后，调用 tcp_handle_recv_data 处理数据：如果 ring_buffer 满了，就需要丢弃该包（但是该实验还没有处理丢包的逻辑，所以这种情况需要控制它不会发生）；如果没满，就调用 write_ring_buffer 将能写入的接收数据写入到 ring_bufer 中，唤醒 tcp_sock_read（wait_recv），将读到 ring_buffer 中的数据读出，在应用内部做进一步的处理。之后修改 rcv_nxt 和 snd_una 的值，发送 ACK 报文。

client 收到 ACK 报文后，会唤醒 tcp_sock_write（wait_send），表示发送完成。随后 client 会接着调用 tcp_sock_read 操作等待要到达的数据（wait_recv）。

server 接下来调用 tcp_sock_write，将刚刚收到的消息带上“server echoes:”的字样全部发送回给 client，然后陷入等待（wait_send）。

client 接收到数据包后，调用 tcp_handle_recv_data 处理数据：如果 ring_buffer 满了，就需要丢弃该包（但是该实验还没有处理丢包的逻辑，所以这种情况需要控制它不会发生）；如果没满，就调用 write_ring_buffer 将能写入的接收数据写入到 ring_bufer 中，唤醒 tcp_sock_read（wait_recv），将读到 ring_buffer 中的数据读出，在应用内部做进一步的处理。之后修改 rcv_nxt 和 snd_una 的值，发送 ACK 报文。之后睡眠 1000 微秒，等待下一轮重复。

server 收到 ACK 报文后，会唤醒 tcp_sock_write（wait_send），表示发送完成。

最后，需要注意的是，收到数据包这一事件与两个过程相关：tcp_sock_read 和 tcp_handle_recv_data。收到一个数据包，首先要经由 tcp_handle_recv_data 处理。正常情况下，收到的数据可以全部放入环形 buf 中，每一次写完后都要唤醒一次 wait_recv（意义为如果有 tcp_sock_read 因环形 buf 为空陷入等待，就将其唤醒）。但是，如果出现了放不进去（buf 满）的情况，就要陷入等待，等 tcp_sock_read 将数据从环形 buf 中读出去后，再次调用 wake_up 唤醒 tcp_handle_recv_data，才能继续将未写入的数据写入。此时需注意，如果在等待写入期间，又有数据包到达，那么该包将丢失。

2.启动脚本

1) TCP server client

```
make all
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# ./tcp_stack server 10001
h2# ./tcp_stack client 10.0.0.1 10001
mininet> quit
```

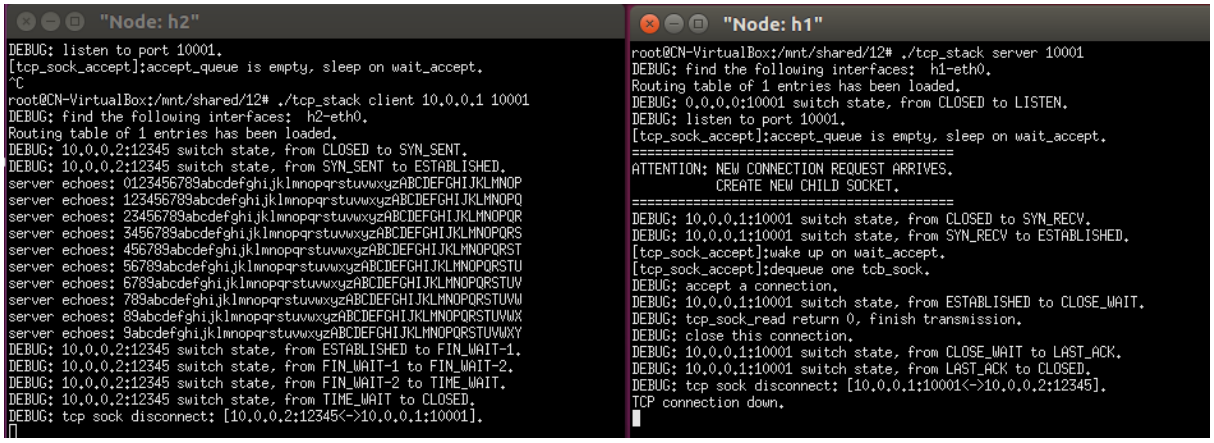
2) TCP server client 文件传输

```
make all
sudo python tcp_topo.py
mininet> xterm h1 h2
h1# ./tcp_stack server 10001
h2# python tcp_stack.py client 10.0.0.1 10001
mininet> quit
```

注：上次实验已经验证状态转移的鲁棒性，故本次不再交叉验证（而且改 py 文件还蛮麻烦，就不改了 233）

三、实验结果及分析

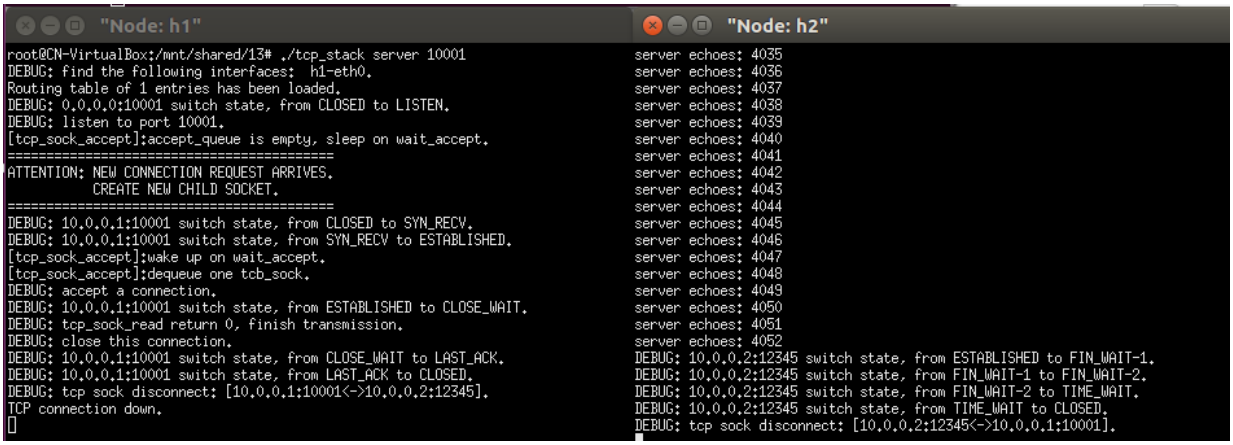
1. 实验结果



图二 TCP server client 实验结果

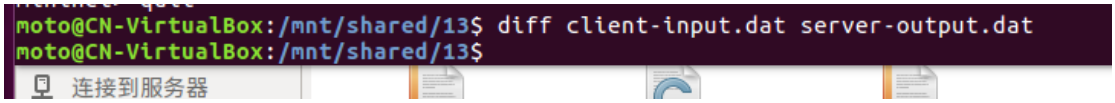
可以看到，在三次握手连接建立完成后，client（h2）发送给 h1 的内容，被 h1 添加“server echoes：”报头后原封不动的返回了过来。传送起始点从“0”到“9”共更改 10 次，也体现在了打印结果上。

在预设的 10 次传输完成后，client（h2）发起关闭连接的请求，server（h1）响应之。整个状态变化过程与上一次实验相同。



图三 TCP server client 文件传输实验结果

这次与之前有所不同的是，文件的内容是 client（h2）打开本地的 client_input.dat 文件读入的，大小为 4052632B，以每个包大小为 1000B 发送给 server（h1）。server（h1）每次收到数据后，将数据写入到本地的 server_output.dat 中，并将计数器加 1 发送给 client（h2）。在传输完成后，双方结束连接。



图四 TCP server client 文件传输比较实验结果

调用 diff 命令比较 client_input.dat 和 server-output.dat，发现完全一致。说明实现了稳定传输。

（二）实验代码详解

本次实验代码过长，不再赘述代码的内容。

一、 tcp_in.c:TCP 状态机具体实现

（1）void tcp_process(struct tcp_sock *tsk, struct tcp_cb *cb, char *packet)：状态响应函数

与上周作业相比，修改了状态响应函数：

如果当前状态为 ESTABLISHED，并且收到数据包不是 FIN 包，意味着要进行数据处理：

如果收到的数据长度为 0，说明只是 ACK 报文，表明自己是消息发送方，此时需要修改 snd_una 为 ack，rcv_nxt 为 seq+1，并且调用 tcp_update_window_safe 更新发送窗口。

否则，说明是数据报文，对方为发送方，此时调用 tcp_handle_recv_data：

首先进入一个循环,循环终止条件为收到的数据包中所有数据都写入到唤醒 buf 中：

检查唤醒 buf 是否已满。如果满，就调用 sleep_on（wait_recv）陷入睡眠。如果未满/被唤醒，就向唤醒 buf 中写入数据（如果能全部写入就全部写入，如果不能就将环形 buf 写满），更新已经写入的数据量，调用 wake_up(wait_recv)唤醒 tcp_sock_read。

修改 recv_nxt 为 seq+len，snd_una 为 ack。

最后发送 ACK。

二、 tcp_sock.c:增加读写函数

（1）int tcp_sock_read(struct tcp_sock *tsk, char *buf, int len)：从环形 buf 中读取数据到应用

首先进入循环：检查环形 buf 是否为空。如果空，就调用 sleep_on(wait_recv)陷入睡眠，等待 tcp_handle_recv_data 唤醒。如果非空，就调用 read_ring_buffer 读取数据，调用 wake_up(wait_recv)唤醒 tcp_handle_recv_data，返回读取数据的长度。

（2）int tcp_sock_write(struct tcp_sock *tsk, char *buf, int len)：将数据发送出去

进入循环，循环结束的条件为剩余待发送长度等于 0：（考虑丢包重传）设置发送长度为：如果当前发送序号+待发送长度-发送确认序号大于默认窗口大小，就设置发送长度为默认窗口大小，否则设为当前发送序号+待发送长度-发送确认序号。调用 tcp_send_data 从发送确认序号标记处开始发送，发送刚刚计算的发送长度的数据，调用 sleep_on（wait_send）陷入睡眠，等待被 tcp_update_window 唤醒，待发送长度减小发送长度的数值。

最后，如果顺利结束，返回 len。

此处部分写的比较绕，主要要考虑的情况太多，仍有不完善的地方。等到下周实验开始写重传时再重新处理这些部分。