实验报告

姓名： 袁峥 学号： 2015K8009929008

一、实验题目：网络传输机制实验三

二、实验内容

1、运行给定网络拓扑(tcp\_topo.py)

2、在节点h1上执行TCP程序

执行脚本(disable\_arp.sh, disable\_icmp.sh, disable\_ip\_forward.sh , disable\_tcp\_rst.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 (./tcp\_stack server 10001)

3、在节点h2上执行TCP程序

执行脚本(disable\_arp.sh, disable\_icmp.sh, disable\_ip\_forward.sh, disable\_tcp\_rst.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并发送数据 (./tcp\_stack client 10.0.0.1 10001)

Client发送文件10MB.dat给server，server将收到的数据存储到文件server-output.dat

4、使用md5sum比较两个文件是否完全相同

三、实验流程

1、网络丢包

带数据或SYN|FIN的包超过一定时间没被确认

①带数据的包没有丢弃，但其对应ACK被丢弃，发送方会认为该包丢失

②网络丢弃ACK数据包，也可能不被双方感知

2、超时重传机制

①每个连接维护一个超时重传定时器

②定时器管理

当发送一个带数据或SYN|FIN的包，如果定时器是关闭的，则开启并设置时间为200ms

当ACK确认了部分数据，重启定时器，设置时间为200ms

当ACK确认了所有数据以及SYN|FIN，关闭定时器

③触发定时器后

重传第一个没有被对方连续确认的数据或SYN|FIN

定时器时间翻倍，记录该数据包的重传次数

当一个数据包重传3次，对方都没有确认，关闭该连接

3、发送队列

①所有未确认的数据或者SYN|FIN包，在收到其对应的ACK之前，都要放在发送队列snd\_buffer（链表实现）中，以备后面可能的重传

②发送新的数据时

放到snd\_buffer队尾，打开定时器

③收到新的ACK

将snd\_buffer中seq\_end <= ack的数据包移除，并更新定时器

④重传定时器触发时

重传snd\_buffer中第一个数据包，定时器数值翻倍

4、接收队列

①数据接收方需要维护两个队列

已经连续收到的数据，放在rcv\_ring\_buffer中供app读取

收到不连续的数据，放到rcv\_ofo\_buffer队列（链表实现）中

②TCP属于发送方驱动传输机制

接收方只负责在收到数据包时回复ACK（收到的最大连续序列号+1）

③收到不连续的数据包时

放在rcv\_ofo\_buffer队列，如果队列中包含了连续数据，则将其移到rcv\_ring\_buffer中

5、超时重传实现

①在tcp\_sock中加入一个新的变量

struct tcp\_timer retrans\_timer;

②当开启定时器时

将retrans\_timer放到timer\_list中

③关闭定时器时

将retrans\_timer从timer\_list中移除

④定时器扫描

建议每10ms扫描一次定时器队列，重传定时器的值为200ms \* 2^N

6、TCP连接建立阶段的丢包处理

①client向server发送的SYN丢包

如果SYN丢了，client在SYN\_SENT状态会超时重传，不会产生问题。

②server向client发送的SYN|ACK丢包

如果server在LISTEN状态回复的SYN|ACK丢包，一方面client会在SYN\_SENT状态超时重传，也就是在server在SYN\_RCVD状态会收到SYN包，收到此包直接丢弃，因为SYN|ACK包本身就会进行超时重传。另一方面，由于该包丢了，因此server在SYN-RCVD状态收不到ACK包，因此会超时重传SYN|ACK包。

③client向server发送的ACK丢包

如果该ACK丢包，也会导致server在SYN\_RCVD状态下重传SYN|ACK包，因此在ESTABLISHED状态会收到SYN|ACK包，回复ACK包即可。但是此时client可能已经进入ESTABLISHED状态并发送了一些数据包，会导致server在SYN\_RCVD状态下收到数据包，如果数据包中含有ACK，则server可以进入ESTABLISHED状态。

7、TCP连接断开阶段的丢包处理

①client向server发送的FIN丢包

如果FIN丢包，client会在FIN\_WAIT\_1状态进行超时重传，不会产生问题。

②server向client发送的ACK丢包

如果ACK丢包，client会在FIN\_WAIT\_1状态超时重传FIN包，也就是说server会在CLOSE\_WAIT状态收到FIN包，比较seq是否正确，并回复ACK即可。同时server甚至此时可能已经发送FIN|ACK包并进入LAST\_ACK状态，这时server会在FIN\_WAIT\_1状态会收到FIN|ACK，可以从中获取ACK信息，并进入FIN\_WAIT\_2状态。

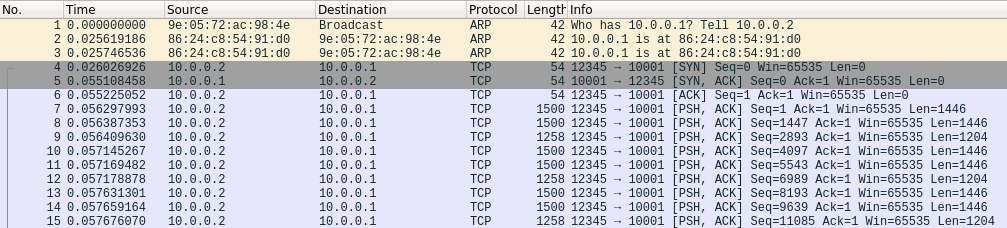
③server向client发送的FIN|ACK丢包

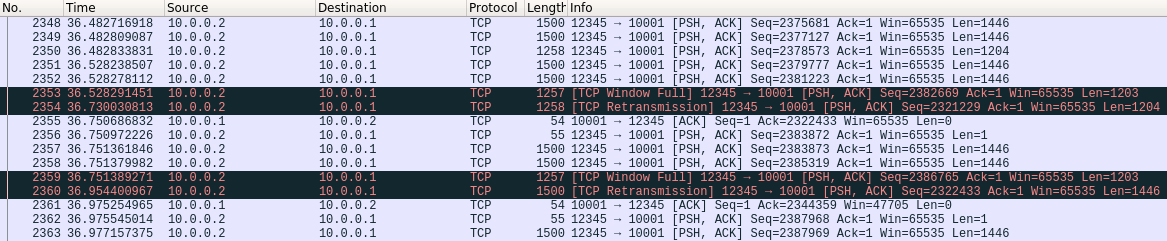
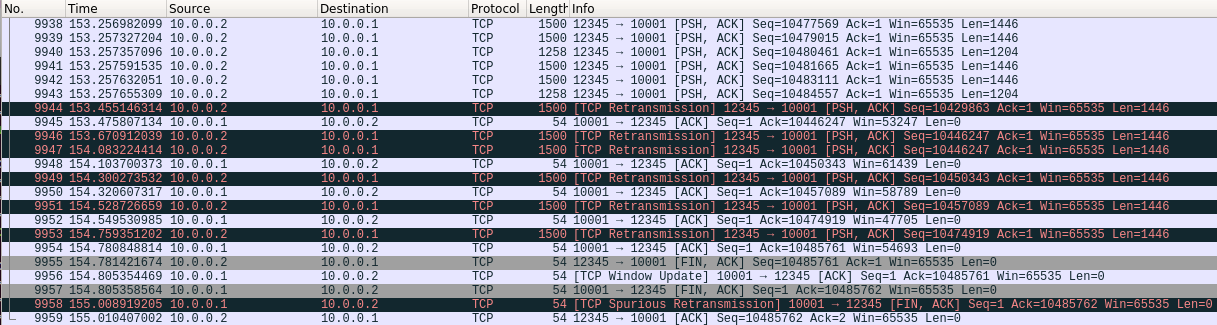
如果FIN|ACK丢包，server会在LAST\_ACK状态超时重传FIN|ACK包，不会产生问题。

④client向server发送的ACK丢包

如果ACK丢包，client在TIME\_WAIT状态下会在2\*MSL时间内收到server发来的FIN|ACK的超时重传，也就是说在client会在TIME\_WAIT状态下收到FIN\_ACK包，此时查看seq和ack是否正确，并且回复ACK，同时重置TIME\_WAIT计时器。

四、实验结果

1、客户端和服务器之间传输10MB文件过程



2、传输过程正常建立和断开连接

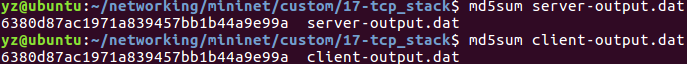
①服务器端



②客户端



3、检查发送文件与接收文件



五、结果分析

从实验结果的1中可以看到，wireshark中清晰地展示了整个传输过程中的建立、数据传输、超时重传、连接断开整个过程，最终顺利完成了10MB文件的传输。

从实验结果的2中可以看到，整个传输过程花费了113秒，此处进行一下说明，在测试时修改了tcp\_topo\_loss.py中的丢包率为10％，同时将超时重传的次数改为了5。以下进行简单计算，如果丢包率为20％，超时重传次数为3，那么无法成功传输一个数据包的概率为20％\*20％\*20％=0.008，而传输10MB文件大约会要花费一万个数据包，从概率来说很可能会出现传输失败的情况，因为为了保证能够顺利传输，将丢包率改为10％，同时超时重传次数改为了5。

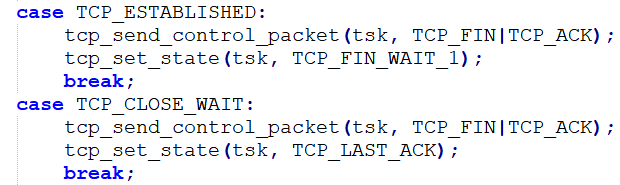
从实验结果的3中可以看到，比较client端发送的文件和server端接收的文件的md5sum值，两者完全一致，说明接收到的文件符合预期。

六、实验总结

这次实验作为本课程的最后一个实验，从难度上来说也是最大的一个，主要的困难在client和server端各自的处理过程中都有两个线程在运行，不同线程间的同步较为复杂，同时收发队列和socket信息在修改时锁的控制也需要注意，还有是各个状态下收到不同的包的处理流程不同，需要注意的细节也比较多，综合以上原因，导致这次实验花费了较长的时间，调试了整整两天才完成任务。下面总结一下此次实验中遇到的问题。

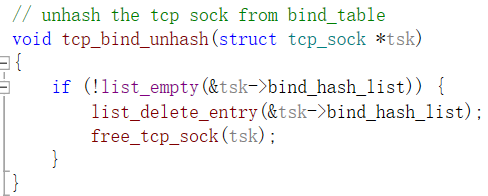
首先是有几个初始代码中的问题。

1、在tcp\_sock.c中的tcp\_sock\_close函数的源代码中，对于TCP\_ESTABLISHED状态和TCP\_CLOSE\_WAIT状态下的处理流程如下：



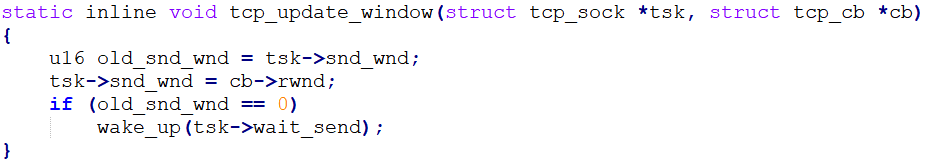
两者均为先发送控制包，再更改当前状态，此处需要把更改状态和发包的先后顺序调换，因为如果先发送控制包后更改状态，可能会在发送控制包后更改状态前，切换至另一线程处理收到的控制包的回复，此时由于状态仍未修改，就会就进入错误状态下的收包处理。

2、在tcp\_sock.c中的tcp\_bind\_unhash函数的源代码中，处理如下：



此处的判断list\_empty(&tsk->bind\_hash\_list)不能起到其应有的效果，该语句本来是为了方式重复bind\_unhash同一个tsk，如果再次bind\_unhash同一个tsk，条件list\_empty也不会满足，原因是list\_empty是判断list->next == list，但是list\_delete\_entry中只有entry->next->prev = entry->prev;和entry->prev->next = entry->next;，并没有entry->next = entry，因此重复删除时不会满足list->next == list的条件，因此entry->next = entry需要加在list\_delete\_entry函数的最后。否则重复bind\_unhash同一个tsk时，仍会进入free\_tcp\_sock，进而出现错误。

3、在tcp\_in.c中的tcp\_update\_window函数中，tsk->snd\_wnd的更新在源代码中如下：



此处直接把cb->rwnd赋值给了tsk->snd\_wnd，在上次的实验中数据传输量小，并不会出现接收窗口满的情况，因此此处并没有出现问题，现在在此次实验中，由于传输数据量很大，从而发现这种处理方法其实是有问题的。tsk->snd\_wnd 应该为 cb->ack + cb->rwnd – tsk->snd\_nxt，具体举例在课堂报告的PPT中。

其次是一些在此次调试过程中遇到的主要问题。

1、在上次实验的函数tcp\_sock\_write中，如果要发送的数据的总长度len小于目前窗口大小，就先进入sleep状态，就等到窗口大小大于发送数据总长度时再唤醒发送，但是这样窗口大小就不会减到0，但是在tcp\_update\_window函数中，发送数据的唤醒条件是老的发送窗口大小为0，因此唤醒条件无法满足。此处发送数据时应该剩余多大的发送窗口就先发送多少数据，将发送窗口大小减至0后再进入睡眠。

2、client在发送完所有数据包后不能立刻发送FIN包进入FIN\_WAIT\_1状态，因为数据包可能还会需要超时重传，应该等到snd\_buffer为空时再发送FIN包。

3、此前的处理中server端如果收到FIN包，则意味着发送方没有新的数据包要发送，则server端唤醒退出tcp\_socket\_read，并进入tcp\_socket\_close，但是此处需要注意，server端在收到FIN包时，不能立刻退出tcp\_socket\_read，还需要注意ring\_buffer中的数据是否全都被读完，如果没有需要等到ring\_buffer中的数据全部被读取后再进入tcp\_socket\_read。

4、数据传输阶段，发送方发送一个数据包，接收方成功接收到后回复ACK包，同时将接收窗口向前滑动，但是如果该ACK丢包，则发送方认为发送不成功，进行先前数据包的超时重传，但此时的数据包已经不在接收方的有效接收窗口内，却仍需回复ACK，不然发送方之前发送的数据包永远也收不到ACK包，会不断进行重传导致错误。

此次需要修改的代码量较大，而且比较分散，很容易就疏忽一些细节，因此需要十分小心。同时我重新组织了tcp\_process函数的处理流程，此前的初始代码中给的提示是根据收到的包的类型不同来进行组织，但是到此次实验中，由于超时重传机制的引入，会导致在一些额外的状态下收到控制包，因此修改起来显得较为混乱。因此我选择根据当前socket的状态不同，进入不同的子函数进行处理，这样逻辑更为清晰，同时修改起来也方便，同时定位查错也更为容易。