**网络传输机制实验一**

范子墨

2021K8009929006

1. **实验内容**
2. 实验内容一：连接管理
   1. 运行给定网络拓扑(tcp\_topo.py)
   2. 在节点h1上执行TCP程序
      1. 执行脚本(disable\_tcp\_rst.sh, disable\_offloading.sh)，禁止协议栈的相应功能
      2. 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
      1. 执行脚本(disable\_tcp\_rst.sh, disable\_offloading.sh)，禁止协议栈的相应功能
      2. 在h2上运行TCP协议栈的客户端模式，连接至h1，显示建立连接成功后自动断开连接 (./tcp\_stack client 10.0.0.1 10001)
   4. 可以在一端用tcp\_stack\_conn.py替换tcp\_stack执行，测试另一端
   5. 通过wireshark抓包来来验证建立和断开连接的正确性
3. 实验内容二：短消息收发
   1. 参照tcp\_stack\_trans.py，修改tcp\_apps.c，使之能够收发短消息
   2. 运行给定网络拓扑(tcp\_topo.py)
   3. 在节点h1上执行TCP程序
      1. 执行脚本(disable\_offloading.sh , disable\_tcp\_rst.sh)
      2. 在h1上运行TCP协议栈的服务器模式 (./tcp\_stack server 10001)
   4. 在节点h2上执行TCP程序
      1. 执行脚本(disable\_offloading.sh, disable\_tcp\_rst.sh)
      2. 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 (./tcp\_stack client 10.0.0.1 10001)
         1. client向server发送数据，server将数据echo给client
   5. 使用tcp\_stack\_trans.py替换其中任意一端，对端都能正确收发数据
4. 实验内容三：大文件传送
   1. 修改tcp\_apps.c(以及tcp\_stack\_trans.py)，使之能够收发文件
   2. 执行create\_randfile.sh，生成待传输数据文件client-input.dat
   3. 运行给定网络拓扑(tcp\_topo.py)
   4. 在节点h1上执行TCP程序
      1. 执行脚本(disable\_offloading.sh , disable\_tcp\_rst.sh)
      2. 在h1上运行TCP协议栈的服务器模式 (./tcp\_stack server 10001)
   5. 在节点h2上执行TCP程序
      1. 执行脚本(disable\_offloading.sh, disable\_tcp\_rst.sh)
      2. 在h2上运行TCP协议栈的客户端模式 (./tcp\_stack client 10.0.0.1 10001)
         1. 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
   6. 使用md5sum比较两个文件是否完全相同
   7. 使用tcp\_stack\_trans.py替换其中任意一端，对端都能正确收发数据
5. **实验内容一：连接管理**
6. 总体代码思路

首先依据原有代码，在main函数中首先调用函数初始化用户态协议栈、路由表和TCP协议栈，在初始化用户态协议栈init\_tcp\_stack函数中，它创建了一个新的线程用于扫描并处理TCP定时器列表（tcp\_scan\_timer\_list函数），这个函数实现了从TIME\_WAIT到CLOSED状态转换。而后调用run\_application函数，传递输入参数后根据创造参数创造新的线程，新线程调用tcp\_server或者tcp\_client函数。main函数中最后调用ustack\_run函数监听网络接口上的数据包并最后调用handle\_packet函数处理，handle\_packet可以跳到handle\_ip\_packet，判断上层协议为TCP后，调用handle\_tcp\_packet进行处理。

在tcp\_server函数中，初始化TCP套接字，调用tcp\_sock\_bind函数将套接字的源端口设置为参数传入的端口号，调用tcp\_sock\_listen函数监听，使用tcp\_sock\_accept函数更改tsk状态并赋给csk，接受连接，最后调用tcp\_sock\_close关闭连接。关闭连接。在TCP\_client函数中，分配TCP套接字，连接服务器，进行数据传输最后关闭连接。

在tcp\_client函数中，初始化套接字，而后调用tcp\_sock\_connect函数，建立连接请求，最后调用tcp\_sock\_close关闭连接。

对于TCP数据包的处理，在handle\_tcp\_packet中首先会初始化一个tcp\_cb的结构体，该结构体中包含了TCP报文的相关信息（如源地址源端口号、目的地址目的端口号等），而后使用tcp\_sock\_lookup函数查找与tcp报文相关联的套接字。最终调用tcp\_process函数处理tcp报文，tcp\_process会根据报文内容和套接字当前状态进行回复，并进行状态转换。

1. 具体实现
2. 信号量的睡眠和唤醒

这是实现线程同步最重要的一步，在套接字结构体定义中含有下列四个信号量。

    struct synch\_wait \*wait\_connect;

    struct synch\_wait \*wait\_accept;

    struct synch\_wait \*wait\_recv;

    struct synch\_wait \*wait\_send;

wait\_connect用于在套接字建立连接之前进行同步，因此在client端发送连接请求后挂起（CLOSED状态变为SYN\_SENT状态），在收到ACK|SYN包之后说明连接client端连接已成功建立（SYN\_SENT状态变为ESTABLISHED状态），因此唤醒该线程。

wait\_accept相对于wait\_connect来说用在server端，当无连接请求时被挂起（处在SYN\_RECV状态），当收到client端发送的ACK包的时候被唤醒（有SYN\_RECV转换到ESTABLISHED状态），此时建立连接的三次招手已经完成。

wait\_recv在接收到的缓冲区为空时被挂起，当收到数据时唤醒。（在实验一中未涉及到）

wait\_send在server端接收窗口大小不足时被挂起，收到PSH|ACK包时进行确认，判断窗口大小是否足够，若足够则唤醒。（在实验一中未涉及到）

1. tcp\_scan\_timer\_list：

该函数对每个定时器的时间进行递减，而后判断是否定时器时间已经归零，若归零，则说明已经超时，则将该定时器从定时器列表中删除，并将对应的套接字状态设置为关闭。

if (timer->timeout <= 0)

        {

            list\_delete\_entry(&timer->list);

            tsk = timewait\_to\_tcp\_sock(timer);

            if (!tsk->parent)

                tcp\_bind\_unhash(tsk);

            tcp\_set\_state(tsk, TCP\_CLOSED);

            free\_tcp\_sock(tsk);

        }

1. tcp\_sock\_listen：设置该套接字能够排队等待连接的最大数量，并且将状态从CLOSED转换为LISTEN。
2. tcp\_sock\_accept：如果等待连接的队列不空，则弹出第一个等待连接的请求并接受。若为空，则挂起该线程。
3. tcp\_sock\_lookup

该函数中首先将提取cb结构体地址中的源端口源地址和目的端口目的地址，而后调用tcp\_sock\_lookup\_established找到与该结构体对应的套接字并返回。具体来说，首先计算哈希值，而后获取对应的哈希表，遍历链表寻找匹配的套接字。

int hash = tcp\_hash\_function(saddr, daddr, sport, dport);

    struct list\_head \* list = &tcp\_established\_sock\_table[hash];

如果寻找到匹配的套接字，则调用tcp\_lookup\_listen函数处理一个新的连接请求，只关注源端口号，在监听套接字哈希表中寻找。

1. tcp\_sock\_connect

这是由client端主动发起的连接请求，向server端发送SYN，而后进行等待。

tcp\_send\_control\_packet(tsk, TCP\_SYN);

    tcp\_set\_state(tsk, TCP\_SYN\_SENT);

    tcp\_hash(tsk);

    sleep\_on(tsk->wait\_connect);

1. tcp\_process

这个函数中是一个状态机的书写。整体状态转换如下图所示，准确来说可以简略成下图。

在这其中，需要注意server端在LISTEN状态收到AYN时，需要建立一个新的子套接字，并将子套接字添加到父套接字的监听队列中，同时将子套接字添加到相应的哈希表中。

                struct tcp\_sock \*child\_sk = alloc\_tcp\_sock();

                child\_sk->parent = tsk;

                child\_sk->sk\_sip = cb->daddr;

                child\_sk->sk\_sport = cb->dport;

                child\_sk->sk\_dip = cb->saddr;

                child\_sk->sk\_dport = cb->sport;

                child\_sk->iss = tcp\_new\_iss();

                child\_sk->rcv\_nxt = cb->seq + 1;

                child\_sk->snd\_nxt = child\_sk->iss;

                list\_add\_tail(&child\_sk->list, &tsk->listen\_queue);

                tcp\_send\_control\_packet(child\_sk, TCP\_ACK | TCP\_SYN);

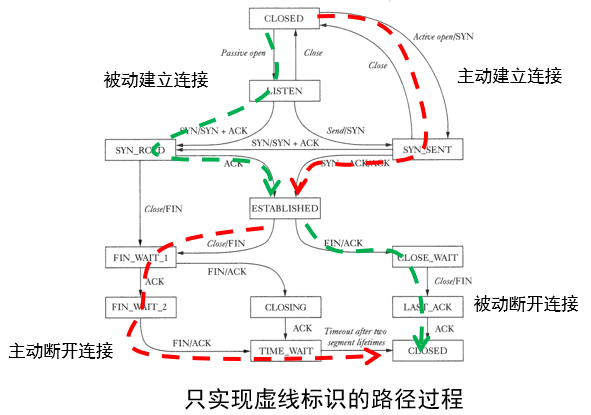
                tcp\_set\_state(child\_sk, TCP\_SYN\_RECV);

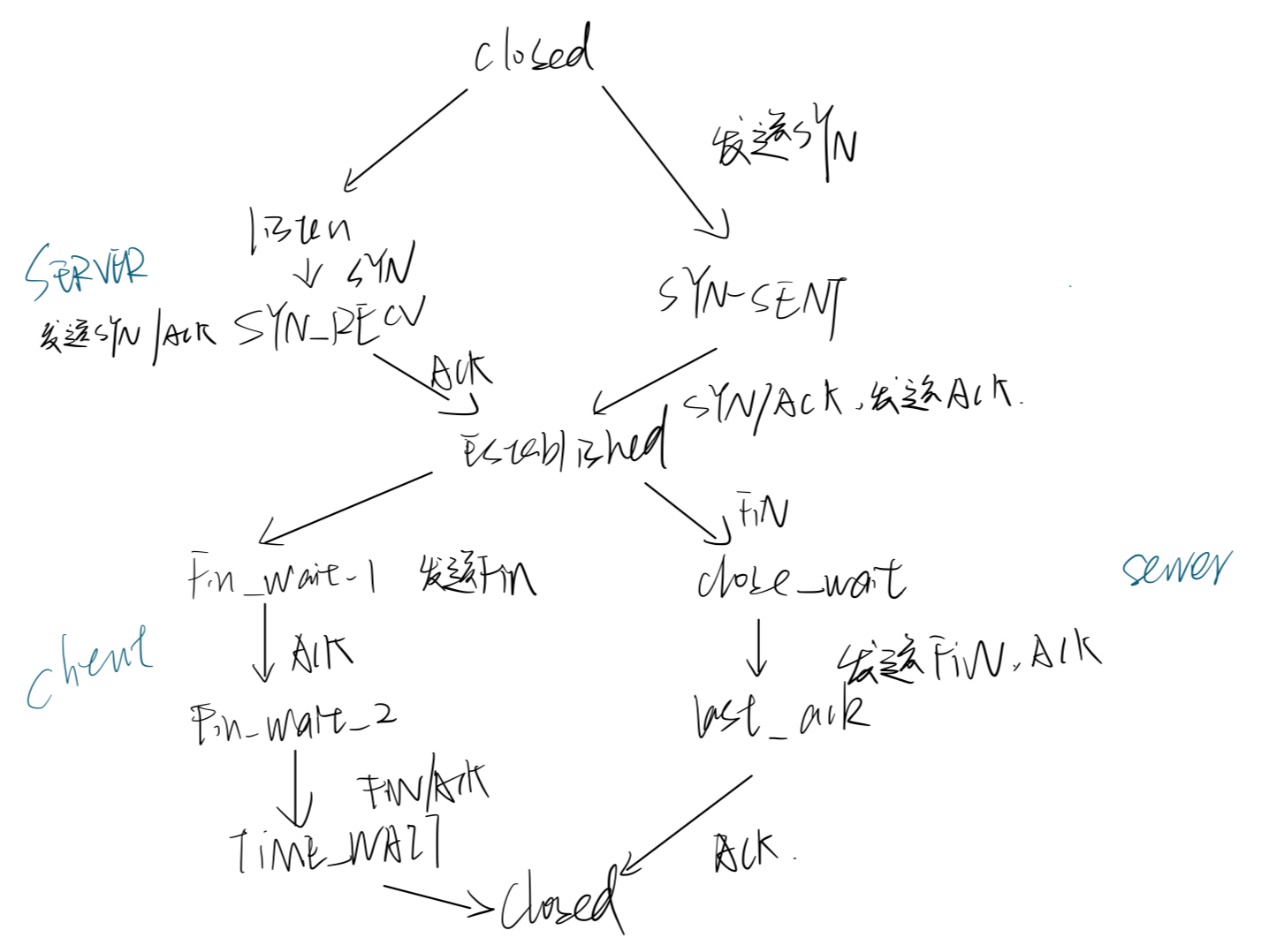
                tcp\_hash(child\_sk);

同时，注意每次更新下一个期望接收到的字节的序列号。信号量的唤醒按照前述的进行添加即可。此外，在TCP\_FIN\_WAIT\_2状态收到FIN包后，会进入TCP\_TIME\_WAIT函数，此时会设置时钟，在超时后自动转换为CLOSED状态。即在tcp\_set\_timewait\_timer函数中将该线程加入定时器列表。

tsk->timewait.timeout = time(NULL);

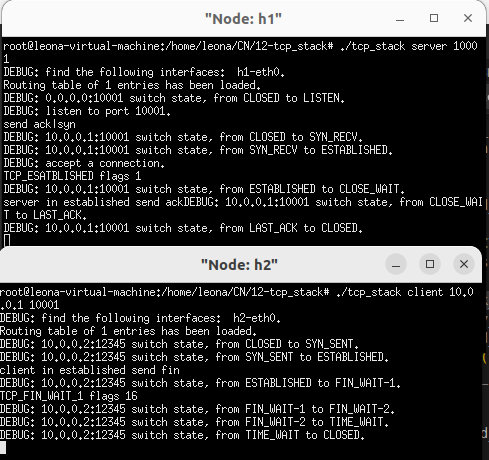
    list\_add\_tail(&tsk->timewait.list,&timer\_list);



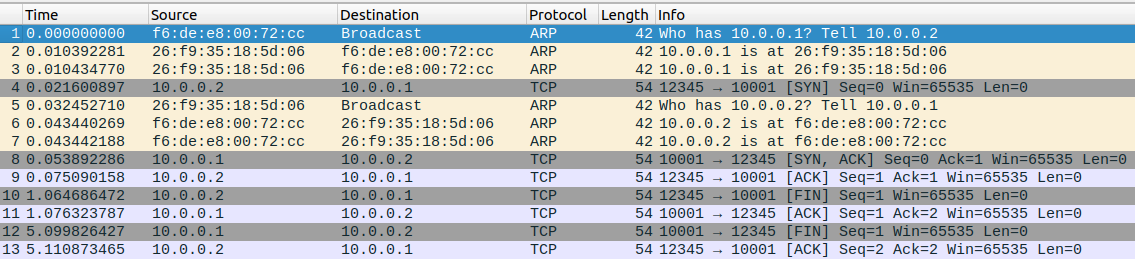


1. 实验结果

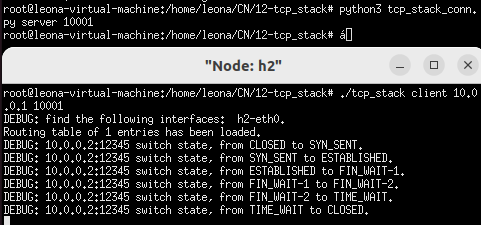
（1）可以成功建立连接，并自动断开连接。

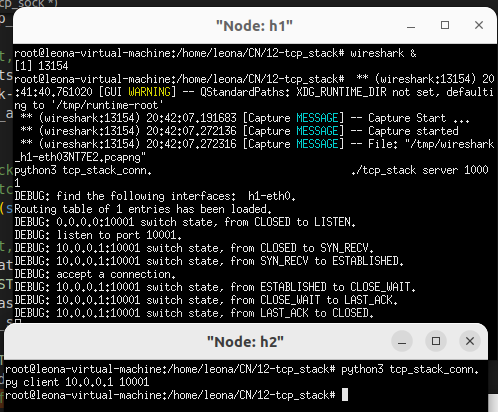


（2）使用wireshark抓包结果如下



（3）在一端用tcp\_stack\_conn.py替换tcp\_stack执行，测试另一端





1. **实验内容二：短消息收发**
2. 具体实现

相比于上个实验，这个实验中要可以收发短消息。总体来说，需要在上个实验的基础上添加状态处理逻辑和读写函数设计。

由老师给出的tcp\_apps.c文件中，在server端，使用tcp\_sock\_read从client端接收数据，通过tcp\_sock\_write向客户端发送数据。在client端，通过tcp\_sock\_write向server端发送数据，使用tcp\_sock\_read从服务器端接收响应数据。与此同时，在tcp\_process函数中添加处理PSH|ACK包的逻辑。

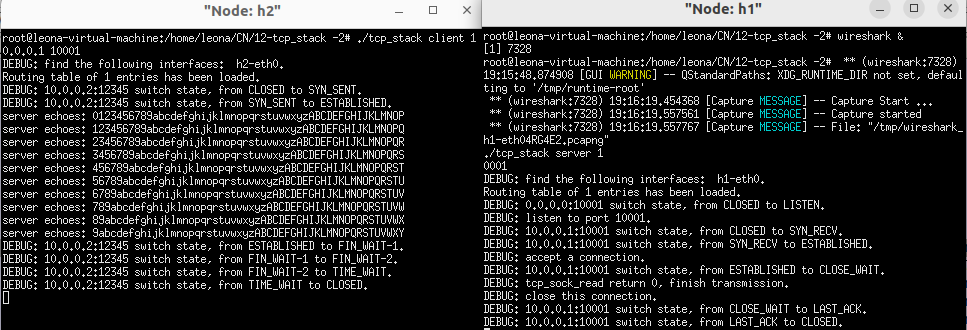
在tcp\_sock\_read函数中，检查缓冲区是否为空，若为空则挂起当前线程。如果收到PSH|FIN包，则说明有数据可读，会被唤醒，则调用read\_ring\_buffer函数读取数据，并将读取长度返回。

在tcp\_sock\_write函数中，将数据分割成适当大小的TCP进行发送。即每次对发送长度进行判断，尽量以TCP包最大容量发送，如果窗口大小不够，则当前进程挂起，在收到ACK包时进行判断后唤醒。

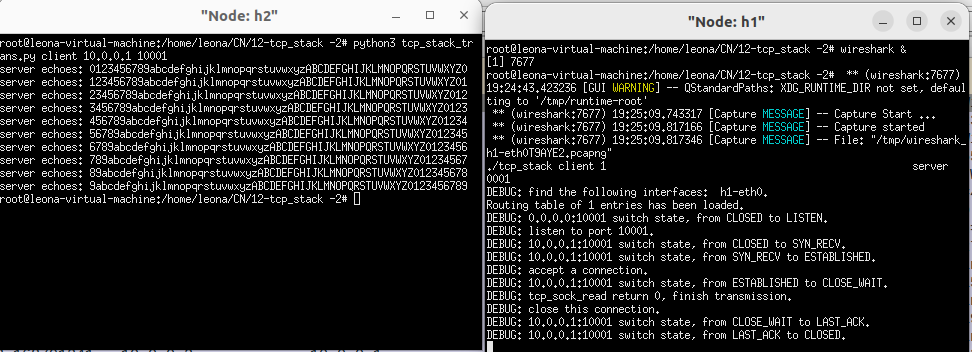
在tcp\_process函数中添加收到PSH|ACK数据包的处理逻辑，将收到的内容写入缓冲区，并恢复ACK包，同时由于出现缓冲区有数据可读，唤醒tcp\_sock\_read所在线程进行读取。

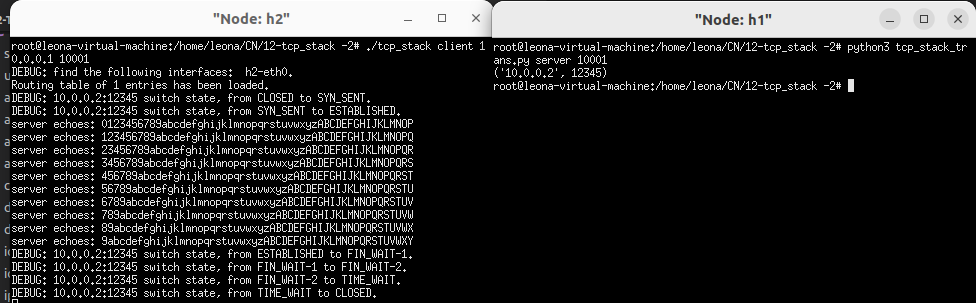
1. 实验结果

（1）节点上运行tcp程序并抓包



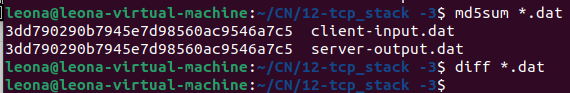
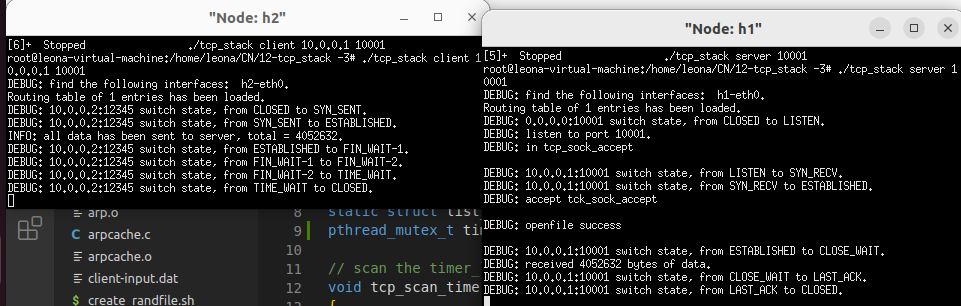
（2）使用tcp\_stack\_trans.py替换其中任意一端



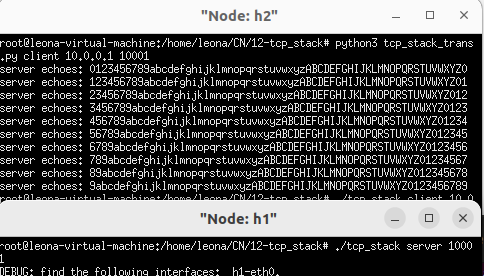


1. **实验内容三：大文件传输**
2. 实验结果

(1)两端均使用自己的代码，可以看出成功传输且传输内容正确。

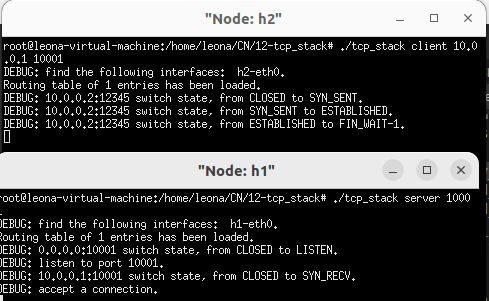


(2)一端使用自己的代码，另一端使用内核版本py，均可以正常运行

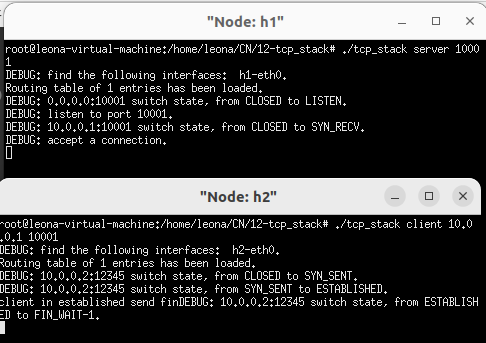


1. **遇到的问题**
2. 状态转换错误

在实验一中发现无法自动断开连接，第一次h2在ESTABLISHED状态停止，返回查看tcp\_sock\_close，发现发送的数据包错误，原来为TCP\_FIN|TCP\_ACK，但实际上应为只有TCP\_FIN，更改后如下图可以进入FIN\_WAIT-1阶段，但依旧卡住。



发现主动连接端（client）成功发送了fin包，但是被动连接端没有收到，因此无法断开连接。



再次进行debug，发现其实server端停在SYN\_RECV后不变化，但是client是成功变化的，因此查看状态机部分，发现在server端收到ACK后，虽然成功进行连接，但是没有进行状态转换，因此要添加状态转换语句。

case TCP\_SYN\_RECV:

            if(cb->flags & TCP\_ACK)

            {

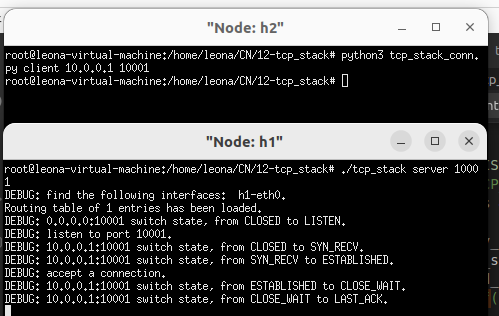
                tcp\_sock\_accept\_enqueue(tsk);

                tsk->rcv\_nxt = cb->seq;

                tsk->snd\_una = cb->ack;

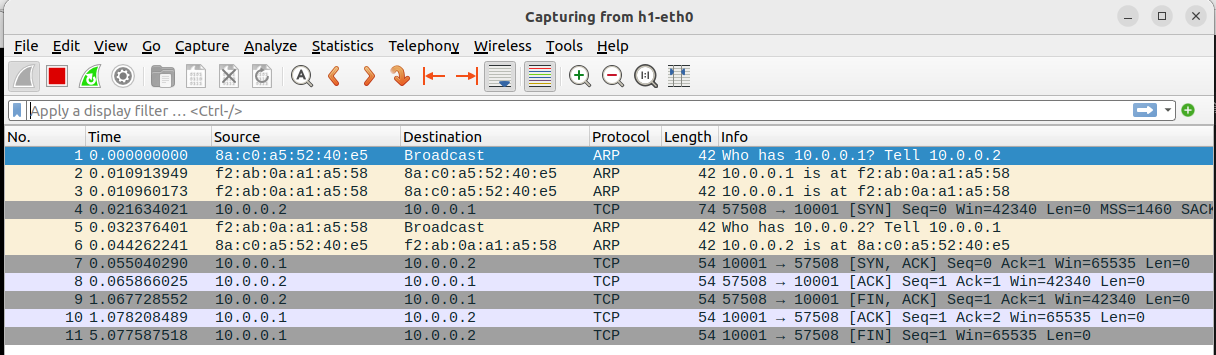
                wake\_up(tsk->parent->wait\_accept);

            }

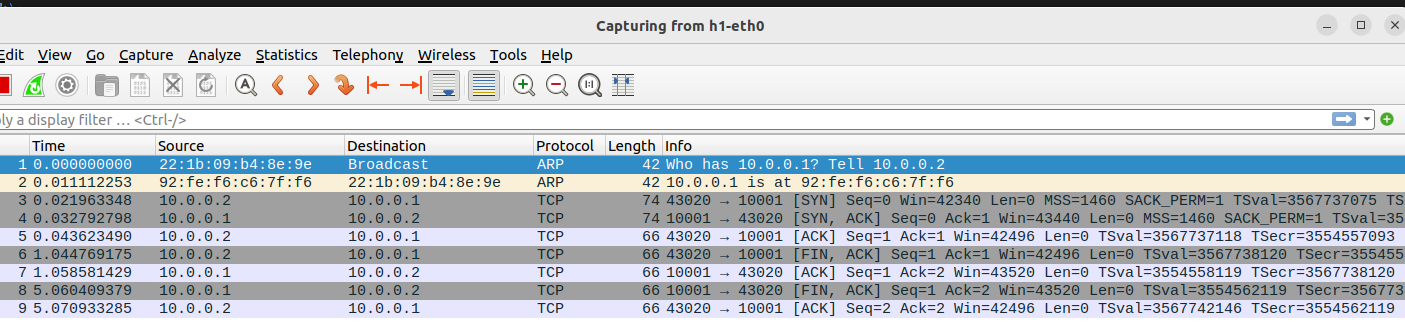


1. 发包错误

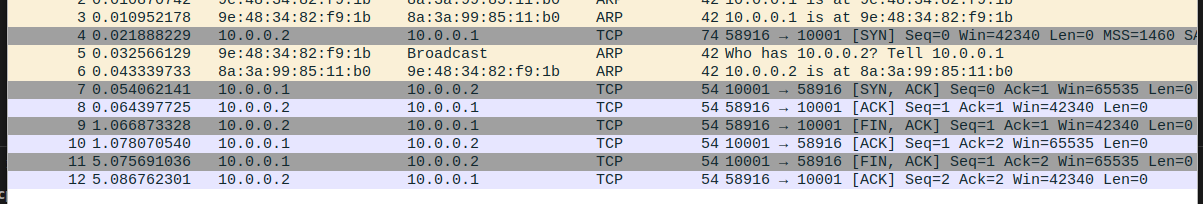
在serve端使用自己的代码，在client端使用内核版本时发现serve端会在状态到达last\_ack的时候停住，无法自己结束。询问助教后，使用wireshark抓包查看，发现最终应由client端向server端发送ack，但是抓包中没有显示。



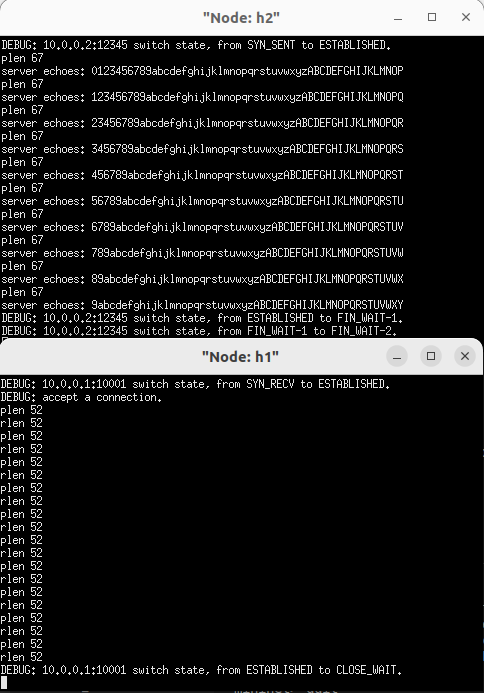
由助教建议，查看都是内核版本的wireshark抓包，发现server端由CLOSE\_WAIT状态转换为LAST\_ACK时发送的是fin|ack包，但我只发送了fin包，因此进行修改。



修改后成功自动关闭连接，实现四次挥手。



3、在实验二中，发现server端会停在close\_wait，client端会停在fin\_wait\_2，根据抓包来看，发现最后client端已经发出了fin包，close\_wait也已经对应发出了ack包，但是最终停止不动。最后发现是在server端收到fin包时，由于此次传输已经结束，因此需要退出所有等待的进程，即调用wait\_exit，将dead位设置为1，并向所有等待的进程广播。



wait\_exit(tsk->wait\_recv);

4、再次无法结束

发现是apps文件中server端无法跳出while循环，仔细查看代码，是由于break的条件是rlen<0，但是当read函数读完之后返回值是0，因此不会跳出循环，换成rlen<=0后即可。

rlen = tcp\_sock\_read(csk, buffer, sizeof(buffer));

if (rlen <= 0)

break;

1. **感想**

由于在上个实验中使用到了哈希表映射的方式，因此这个实验中哈希表的使用比较得心应手，不会感觉到陌生。在实验中，对于线程同步的理解加深了许多。以及本次实验中由于三次实验的内容不同，因此对于自己要写什么有些许迷茫，老师给出的apps.c文件中也有小小的问题。另一个难点在于老师其实给出了很多已经实现的功能封装在函数中，但是就像散落的拼图碎片，需要自己理解后并在实验中拼起来，有点困难。整个实验算得上是到现在最为艰难的实验，耗费了很长的时间。在本地运行的时候，可能由于线程之间的同步完成的不是很好，导致代码有时能跑出结果有时候不能，让本就艰难的实验雪上加霜。