网络传输机制实验三报告

李昊宸

2017K8009929044

（一）TCP可靠传输实现

一、实验内容

1. TCP server client文件传输实验：

1）运行网络拓扑(tcp\_topo\_loss.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

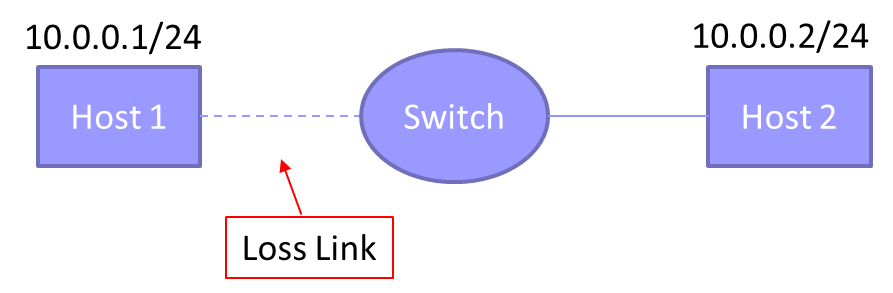
tcp\_sock.c ： tcp\_sock操作相关函数

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\_loss.py： 实现二节点的丢包率为2%的拓扑



图一 丢包率为2%的二节点网络拓扑

上周的实验已经实现TCP的稳定传输，这周需要补充在连接建立后进行数据的可靠传输。

TCP的超时重传机制：为每个连接维护一个超时重传定时器。

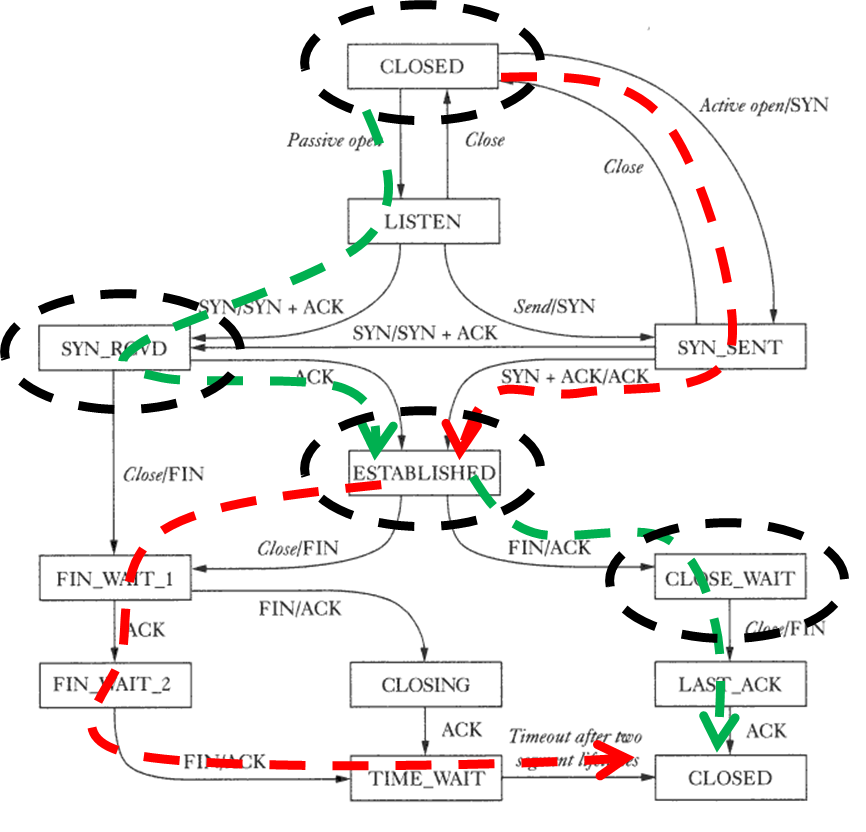
1）定时器管理

当发送一个带数据/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/SYN/FIN，关闭定时器。

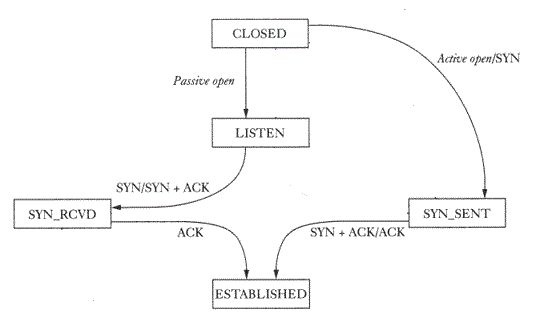
2）触发定时器后

重传第一个没有被对方连续确认的数据/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(RST)。

TCP可能出现发送的包丢包的位置：



连接建立（有丢包）：



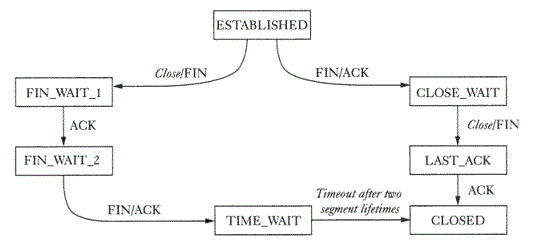
主动建立连接的场合：

1. 发送SYN，该数据包被丢弃：主动方状态为SYN\_SENT,被动方为LISTEN，主动方超时后需要重新发送SYN包。
2. 被动方返回的SYN|ACK被丢弃：主动方状态为SYN\_SENT,被动方为SYN\_RCVD，被动方超时后需要重新发送SYN|ACK包。

被动建立连接的场合：

1. 被动方返回的SYN|ACK被丢弃：主动方状态为SYN\_SENT,被动方为SYN\_RCVD，被动方超时后需要重新发送SYN|ACK包。
2. 主动方返回的ACK被丢弃：主动方状态为ESTABLISHED,被动方为SYN\_RCVD，主动方超时后需要重新发送ACK包。

连接释放（有丢包）：



主动关闭连接的场合：

1. 发送FIN|ACK，该数据包被丢弃：主动方状态为FIN\_WAIT\_1,被动方为ESTABLISHED，主动方超时后需要重新发送FIN|ACK包。
2. 被动方返回的ACK被丢弃：主动方状态为FIN\_WAIT\_1,被动方为CLOSE\_WAIT，被动方超时后需要重新发送ACK包。

被动关闭连接的场合：

1. 被动方发送的FIN|ACK被丢弃：主动方状态为FIN\_WAIT\_2,被动方为LAST\_ACK，被动方超时后需要重新发送FIN|ACK包。
2. 主动方返回的ACK被丢弃：主动方状态为TIME\_WAIT->CLOSED,被动方为LAST\_ACK，主动方超时后需要重新发送ACK包。

TCP发送队列：

所有未确认的数据/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。

1）发送新的数据时

放到snd\_buffer队尾，打开定时器（将对应tsk放入扫描队列中）。

2）收到新的ACK

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

3）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。当重传次数超过三次时，断开连接。

4）切换状态时

建议关闭定时器（将对应tsk从扫描队列中删除），等到再次发送数据时重新开始定时器。

5）定时器扫描

每10ms扫描一次定时器队列。

TCP接收队列：

数据接收方需要维护两个队列

1）已经连续收到的数据，放在rcv\_ring\_buffer中供app读取

2）收到不连续的数据，放到rcv\_ofo\_buffer队列（链表实现）中

接收方负责在收到数据包时回复相应ACK，收到不连续的数据包时，按序放在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。

2.启动脚本

1) TCP server client文件传输

make all

sudo python tcp\_topo\_loss.py

mininet> xterm h1 h2

h1# ./tcp\_stack server 10001

h2# python tcp\_stack.py client 10.0.0.1 10001

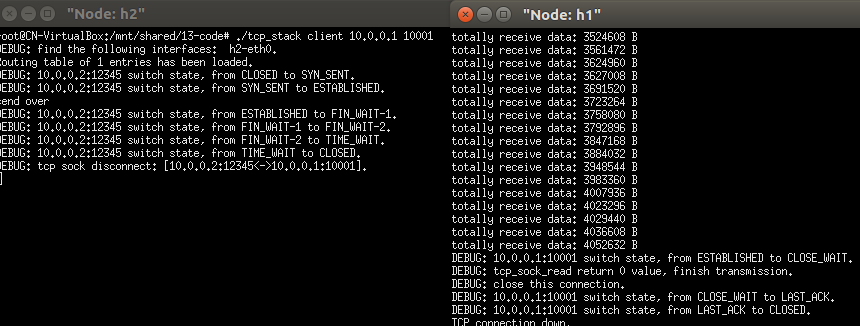
mininet> quit

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）

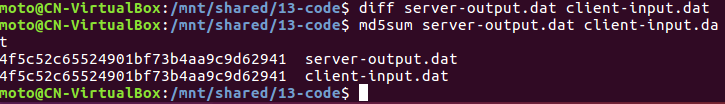
三、实验结果及分析

1. 实验结果

图二 TCP可靠传输实验结果

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，在h1保存完之后，打印当前所收到的总比特数。

在数据传输完成后（达到了文件大小4052632 B），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可靠传输文件比较实验结果

调用diff命令比较client\_input.dat和server-output.dat，发现完全一致。

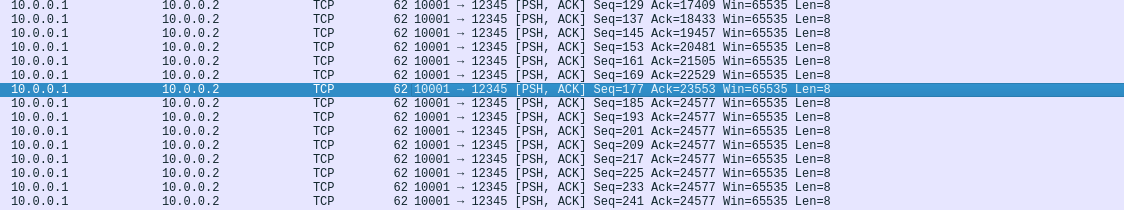
调用md5sum命令比较client\_input.dat和server-output.dat，发现完全一致。

说明实现了稳定传输。



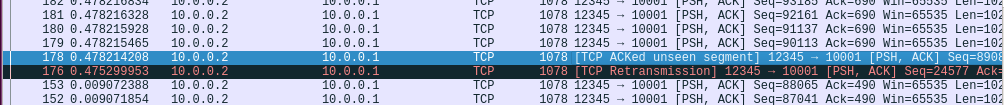
图四 wireshark抓取丢失包

在h1节点的eth0端口进行抓包，发现在每个包大小为1024B的基础上，序列号为23553的包与序列号为25601的包中间的序列号为24577的包在传输过程中丢失。



图五 wireshark抓取应答包

可以发现，在h1向h2发送的确认报文中，从24577开始序列号不再增长，说明接收方没有收到序列号24577的包，不再进行继续确认。



图六 wireshark抓取丢失包

重传计时器超时的h2，重新发送了序列号为24577的包，并被h1所接收。

以上表明了链路在具有丢包的情况下，传输仍然可以稳定进行，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。另外，本次实验较难，部分处理逻辑询问了认识的学长的处理模式（如对乱序到达队列ofo的处理）。

1. tcp\_sock.h:新增关于发送缓冲区与接收缓冲区的数据结构

（1）发送缓冲区：

struct tcp\_send\_buffer\_block{

struct list\_head list;

int len; //包的总长度

char\* packet;

};

每一个块表示在每一次调用tcp\_send\_packet时发送的包的缓存，len记录packet的总长度。

struct tcp\_send\_buffer{

struct list\_head list;

int size;

pthread\_mutex\_t lock;

pthread\_t thread\_retrans\_timer;

} send\_buffer;

每一个缓存块通过调用list\_add\_tail或list\_insert按序列号插入到list上，timer为重传计时器，lock为访问数据区的互斥锁。

（2）乱序接收缓冲区：

struct tcp\_ofo\_block {

struct list\_head list;

u32 seq;

u32 len; //数据部分的总长度

char\* data;

};

每一个块表示在乱序收到发送的包时，将包解析缓存下来的数据，并且按照seq的顺序进行有序排列。seq表示收到的包的序列号，len表示收到的包的 数据段的长度，data表示收到的数据。

1. tcp\_in.c:修改发送逻辑

（1）void tcp\_process(struct tcp\_sock \*tsk, struct tcp\_cb \*cb, char \*packet)：对发送逻辑进行修改

如果当前状态为TCP\_LISTEN:

在发送SYN|ACK的控制包前，调用tcp\_set\_retrans\_timer启动重传计时器。

如果当前状态为TCP\_SYN\_SENT：

由于之前发送出去SYN包，如果收到了新的包，并且该包为SYN|ACK，那么说明之前发送的SYN包没有丢失，调用send\_buffer\_ACK处理发送缓存，调用tcp\_unset\_retrans\_timer关闭定时器，状态切换到ESTABLISHED。随后发送一个ACK控制包（注意，该控制包可以不用开启定时器，因为在下一次发送数据时，必将发送一个ACK包，就算该控制包丢失也是不可见的）。

如果当前状态为TCP\_SYN\_RECV：

在收到ACK包后，说明之前发送的SYN|ACK包没有丢失，于是可以调用send\_buffer\_ACK处理发送缓存，调用tcp\_unset\_retrans\_timer关闭重传计时器。

如果当前状态为ESTABLISHED，并且收到的包不是FIN：

如果收到一个单纯的确认ACK或者一个ACK仅带有“data\_recv！”的数据（此处需要进行说明的是，通过对tcp\_send\_control\_packet的修改，在发送控制包时，如果发送到包是一个单纯的ACK包或者为一个单纯的RST包，那么不对其进行缓存，因为不包含数据。除此之外发送的包都进行缓存。这导致一个问题，后去如果出现连续的单纯ACK包丢失的话，数据发送方就会陷入长期的等待，而接收方也没有新的报文需要发送，很可能陷入死锁。所以为避免这种情况，接收方在进行确认时，也需要向发送方发送一个带有数据的ACK，数据规定为“data\_recv！”，并没有实际意义，仅作为一个ACK的识别符，并且如果该ACK报文丢失的话，因为其带有数据，会被缓存并重传，避免了死锁。）说明该方为数据发送方，在更新发送窗口后，需要调用send\_buffer\_ACK处理发送缓存，调用tcp\_update\_retrans\_timer更新重传计时器。

如果当前状态为TCP\_FIM\_WAIT\_1，并且收到的包不是FIN：

说明发送的FIN|ACK到达，调用send\_buffer\_ACK处理发送缓存，调用tcp\_unset\_retrans\_timer关闭重传计时器。

如果当前状态为CLOSING，并且收到的包不是FIN：

说明发送的FIN|ACK到达，调用send\_buffer\_ACK处理发送缓存，调用tcp\_unset\_retrans\_timer关闭重传计时器。

如果当前状态为TCP\_LAST\_ACK，并且收到的包不是FIN：

说明发送的FIN|ACK到达，调用send\_buffer\_ACK处理发送缓存，调用tcp\_unset\_retrans\_timer关闭重传计时器。

（2）void tcp\_recv\_data(struct tcp\_sock \*tsk, struct tcp\_cb \*cb, char \*packet)：对接收逻辑进行修改

如果收到的seq小于rcv\_nxt，就丢弃该包。否则，调用ofo\_packet\_enqueue插入乱序到达缓冲区，调用ofo\_packet\_dequeue将乱序到达缓冲区的顺序部分读出到环形buffer中，并唤醒tcp\_sock\_read读取环形buffer中的数据（注意读完后会关闭上一次开启的重传计时器）。最后调用tcp\_set\_retrans\_timer再次打开重传计时器，发送ACK（含数据data\_recv!）报文。

1. tcp\_sock.c:修改读写函数与缓冲区操作函数

（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，最后调用tcp\_unset\_retrans\_timer关闭本次传输的计时器，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

发包之前，先调用tcp\_set\_retrans\_timer打开重传计时器。

进入循环，循环结束的条件为剩余待发送长度等于0：设置发送长度为剩余待发送长度与默认发送长度（1024）中的最小值。当send\_buffer的大小加上本次发送长度大于发送窗口时，需要陷入睡眠等待（wait\_send），直到小于等于发送窗口时，调用tcp\_send\_data将本次数据发送出去。

从循环出来后，意味着所有数据都已经发送出去。现在等待所有数据都被确认（wait\_send）后，调用tcp\_unset\_retrans\_timer关闭重传计时器。

（3）void tcp\_sock\_close(struct tcp\_sock \*tsk)：修改连接关闭函数

如果当前状态为TCP\_ESTABLISHED:

发送控制包FIN|ACK前，调用tcp\_set\_retrans\_timer打开重传计时器。

如果当前状态为TCP\_CLOSE\_WAIT：

发送控制包FIN|ACK前，调用tcp\_set\_retrans\_timer打开重传计时器。

（4）void send\_buffer\_free()：三次重传后仍未收到ACK，就将发送缓冲区清空，释放连接

（5）void send\_buffer\_RETRAN\_HEAD(struct tcp\_sock \*tsk)：重传send\_buffer中第一个数据包

首先list\_empty查看链表是否为空。

如果非空，就调用list\_entry访问链表的第一个乱序块，将乱序块装包，修改ack为htonl（rcv\_nxt），修改tcp和ip校验和，最后调用ip\_send\_packet发送。

（6）void send\_buffer\_ADD\_TAIL(char\* packet,int len)：发送新的数据时，将数据放到send\_buffer队尾

首先malloc分配一个发送缓冲块。

随后修改len为数据包的大小，packet为包的数据。

最后获取发送缓冲区互斥锁，将send\_buffer的size增加TCP数据包的数据部分长度的大小，将缓冲块调用宏list\_add\_tail增添到链表尾部，释放互斥锁。

（7）void send\_buffer\_ACK(struct tcp\_sock \*tsk, u32 ack)：收到ACK后，将send\_buffer中序列号小于ack的数据包删除

调用list\_for\_each\_entry\_safe安全遍历每个发送缓冲块：解析缓存报头，获得seq。如果seq小于收到ACK报文的ack号，说明该缓存已经顺利被对端接收，可以删除，就获取互斥锁，将send\_buffer的size减小TCP数据包的数据部分长度的大小，将缓冲块调用宏list\_delete\_entry从链表删除，释放缓冲块的空间，释放互斥锁。

（8）void ofo\_packet\_enqueue(struct tcp\_sock \*tsk, struct tcp\_cb \*cb, char \*packet)：乱序到达的packet有序进入优先级缓存队列

新建一个乱序块，读取解析好的tcp\_context\_block，设置乱序块的seq为cb->seq，len为cb->pl\_len，data为收到包的数据段。调用list\_for\_each\_entry\_safe安全遍历乱序缓冲区，调用less\_than\_32b与当前遍历到的块进行序列号比较，如果新建的乱序块的seq更小，就调用list\_add\_tail将其插入。如果遍历结束仍没有插入，就将其加入链表尾部。

（9）int ofo\_packet\_dequeue(struct tcp\_sock \*tsk)：有序化的乱序到达缓冲区向环形buffer输出有序块

设置当前要接收的seq为rcv\_nxt，调用list\_for\_each\_entry\_safe遍历乱序缓冲区：

如果seq与当前遍历到块的seq相等，说明是连续块。当该块的len大于环形buffer的空闲区域时，就打印“sleep on buff\_full”，陷入睡眠（wait\_recv）。等到被read唤醒后，打印“wake up”，继续进行比较。直到len小于空闲区域后，调用write\_ring\_buffer将数据写入缓冲区，调用wake\_up(wait\_recv)唤醒tcp\_sock\_read，设置当前要接收的seq为seq+len，设置rcv\_nxt为seq，调用list\_delete\_entry删除该节点，释放空间，继续遍历。

如果seq与当前遍历到块的seq不相等，说明不连续，结束扫描。

1. tcp\_timer.c:重传计时器

（1）void tcp\_set\_retrans\_timer(struct tcp\_sock \*tsk)：启动重传计时器

设置typr为1（retrans），timeout为MIN\_RETRANS\_TIME（200ms），初始化重传次数为0，调用list\_add\_tail将restrans\_timer->list添加到重传链表上。最后将ref\_cnt加1。

（2）void tcp\_update\_retrans\_timer(struct tcp\_sock \*tsk)：更新重传计时器

设置typr为1（retrans），timeout为MIN\_RETRANS\_TIME（200ms），初始化重传次数为0。

（3）void tcp\_unset\_retrans\_timer(struct tcp\_sock \*tsk)：关闭重传计时器

调用list\_delete\_entry将retrans\_timer从重传链表上删除，调用free\_tcp\_sock减引用。

（4）void \*tcp\_retrans\_timer\_thread(void \*arg)：重传计时器线程

每10ms执行一次tcp\_scan\_retrans\_timer\_list，没啥好说的。

（5）void tcp\_scan\_retrans\_timer\_list()：扫描函数

遍历重传链表：

timeout减少TCP\_RETRANS\_SCAN\_INTERVAL（10ms），调用retrans\_timer\_to\_tcp\_sock找到该计时器所在的tcp\_socket。

如果超时，先检查重传次数：

如果重传次数超过3次，就将该计时器从链表上删除，如果tcp\_socket为父socket，就调用tcp\_bind\_unhash解绑定，调用wait\_exit(tsk->wait\_connect)，wait\_exit(tsk->wait\_accept)，wait\_exit(tsk->wait\_recv)，wait\_exit(tsk->wait\_send)解除所有睡眠，设置状态为TCP\_CLOSED，释放socket，调用send\_buffer\_free释放缓冲区。

如果重传次数没有超过三次，就将retrans\_times加1，timeout设置为10ms\*2^retrans\_times，调用send\_buffer\_RETRAN\_HEAD进行重传。

如果没超时，就忽略。