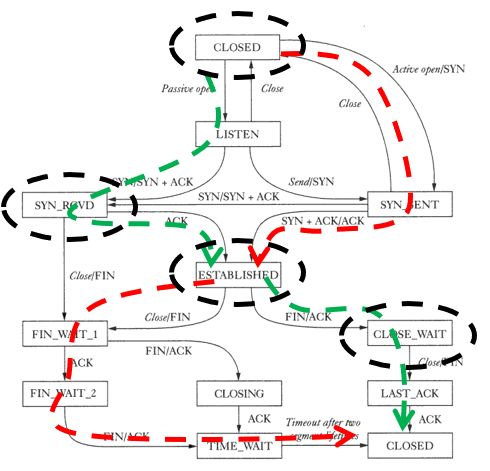
**网络机制实验二**

范子墨

2021K8009929006

1. 实验内容
   1. 执行create\_randfile.sh，生成待传输数据文件client-input.dat
   2. 运行给定网络拓扑(tcp\_topo\_loss.py)
   3. 在节点h1上执行TCP程序
      1. 执行脚本(disable\_offloading.sh , disable\_tcp\_rst.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)
   4. 在节点h2上执行TCP程序
      1. 执行脚本(disable\_offloading.sh, disable\_tcp\_rst.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 (./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
   5. 使用md5sum比较两个文件是否完全相同
   6. 使用tcp\_stack.py替换两端任意一方，对端都能正确处理数据收发
2. 设计思路
3. 总体设计思路

在上一次的实验基础上实现可靠传输，节点之间在有丢包网络中能够建立连接并正确传输数据。当发送数据包时，要将数据包放入send\_buf缓冲区，即对于所有需要发送数据包时候，将数据包放入send\_buf中，即调用new\_data\_block函数后将该结构加入到队列中，并设置定时器。在收到ACK后，调用tcp\_free\_send\_buf函数检查ack大小，删除send\_buf队列中已经确认过的数据包，并删除定时器。当接收数据包时，检查是否符合当前期望序列号，若符合，则唤醒wait\_recv对应线程，并写入接收缓冲区，否则加入rcv\_ofo\_buf缓冲区中。



1. send\_buf队列维护

首先是给send\_buf和rcv\_ofo\_buf重新定义一个新的数据结构，将所有数据包形成链表。数据结构中需要包含flags，发送计数，长度和结束序列号以及数据包内容。

struct data\_packet {

    struct list\_head list;

    u8  flags;

    u8  times;

    u32 seq;

    u32 len;

    u32 seq\_end;

    char \*packet;

};

而后对于所有需要发送数据包的地方进行修改，将数据包加入发送队列。即调用new \_data\_block函数。该函数有四个参数，flags主要是用于判断是否为SYN或者FIN包，seq是序列号，len和buf为数据包的传输数据长度和内容。

struct data\_packet \*new\_ data\_block(u8 flags, u32 seq, u32 len, char \*buf)

在该函数中，会构建一个data\_packet结构体，将该结构体中的变量赋值后返回。这里需要注意结束序列号中要考虑到SYN或者FIN标志。如果是SYN或者FIN的话，由于他们虽然不携带数据（但是内核版本的FIN包是携带数据的），但是TCP数据流的一部分，因此占用一个序列号，结束序列号要加一。

dp->seq\_end = seq + len + ((flags & (TCP\_SYN|TCP\_FIN)) ? 1 : 0);

在tcp\_sock\_connect函数中发送SYN，因此需要将该包加入发送队列中，不包含数据。

struct data\_packet \*dp = new \_data\_block(TCP\_SYN, tsk->snd\_nxt, 0, NULL);

pthread\_mutex\_lock(&send\_buf\_lock);

list\_add\_tail(&dp->list, &tsk->send\_buf.list);

pthread\_mutex\_unlock(&send\_buf\_lock);

在tcp\_sock\_close中同样发送FIN|ACK包，将其加入发送队列中，这里不再赘述。

在tcp\_sock\_write函数中，由于非SYN或者FIN包，不妨将flags设为ACK，其中注意pt为已经发送数据的长度，因此buf+pt即将指针偏移至当前要发送的数据包的起始点。

struct data\_packet \*dp = new\_data\_block(TCP\_ACK, seq, data\_len, buf + pt);

其余发送数据包的地方同理。

当收到ACK后，需要将发送队列中对应的数据包删除。即在tcp\_process函数中对应收到ACK包后，调用tcp\_free\_send\_buf函数。在该函数中，遍历发送队列，比较当前数据包的序列号和发送队列中的结束序列号，如果结束序列号在当前数据包的序列号之前，说明该数据块已经收到ACK成功被接收了，则删除该条目，同时取消重传定时器。由于部分数据已经被确认，则重启定时器，设置时间为200ms。

list\_for\_each\_entry\_safe(tmp, q, &tsk->send\_buf.list, list)

    {

        if(tmp->seq\_end <= cb->ack)

        {

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

            free(tmp->packet);

            free(tmp);

            tcp\_unset\_retrans\_timer(tsk);

            if(!list\_empty(&tsk->send\_buf.list))

                tcp\_set\_retrans\_timer(tsk);

        }

    }

1. rcv\_ofo\_buf队列维护

根据wireshark抓包来看，client端发送数据包为PSH|ACK包，因此可以将对于数据包的处理同样放入对ACK包的处理中。如果期望接收到的序列号与当前收到的序列号，则调用tcp\_rcv\_ofo\_pkt函数尽心处理，并回复ACK。

if(tsk->rcv\_nxt != cb->seq)

                {

                    tcp\_rcv\_ofo\_pkt(tsk, cb);

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

                    break;

                }

在tcp\_rcv\_ofo\_pkt函数中，同样调用new\_data\_block函数构建一个data\_packet结构体，并将该条目插入序列号对应的位置，即乱序接收队列是按照序列号从小到大排列的。

list\_for\_each\_entry\_safe(tmp, q, &tsk->rcv\_ofo\_buf.list, list)

    {

        if(tmp->seq\_end > cb->seq\_end)

            break;

    }

而收到数据包后，如果满足接收到的序列号与期望收到的序列号想等，则遍历乱序接收队列，如果有符合期望接收到的下一个序列号等于条目对应的序列号的（此时期望接收到的序列号已经更新），则将该条目的数据包写入环形缓冲区，并删除该序列号。这里处理的一个基本逻辑是，每个数据包的长度是固定的，即发送内容如何分割是固定的，因此可以直接有序列号与期望收到的序列号是否相等判断，而不需要进行大小判断。此外，定义fin\_flag，即如果同时是FIN包，则发送ACK回复。（内核版本的代码会出现FIN包中包含数据的情况）

list\_for\_each\_entry\_safe(tmp, q, &tsk->rcv\_ofo\_buf.list, list)

                {

                    if(tmp->seq == tsk->rcv\_nxt)

                    {

                        wake\_up(tsk->wait\_recv);

                        pthread\_mutex\_lock(&rcv\_buf\_lock);

                        write\_ring\_buffer(tsk->rcv\_buf, tmp->packet, tmp->len);

                        tsk->rcv\_wnd -= tmp->len;

                        pthread\_mutex\_unlock(&rcv\_buf\_lock);

                        tsk->rcv\_nxt = tmp->seq\_end;

                        fin\_flag = tmp->flags & TCP\_FIN;

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

                        free(tmp->packet);

                        free(tmp);

                    }

                    else

                        break;

                }

1. 定时器维护

与维护发送队列类似，所有发送数据包的地方同样需要设置定时器，即如果没有启用重传定时器，则调用tcp\_set\_retrans\_timer函数启动重传定时器。

if(!tsk->retrans\_timer.enable)

            tcp\_set\_retrans\_timer(tsk);

tcp\_set\_retrans\_timer函数中，设置类型和enable为1，超时时间为TCP\_RETRANS\_INTERVAL\_INITIAL，并将其加入到定时器队列中，注意这里的定时器队列与TIME\_WAIT定时器队列相同，只需在tcp\_scan\_timer\_list中加以区别后处理即可。

tcp\_scan\_timer\_list函数中，遍历定时器列表，在原来的基础上添加判断type=1的情况，如果type=1，如果重传次数已经超过三次，则发送RST包。其余情况按照原来发送数据包的方式即可。

if(dp->flags & (TCP\_SYN | TCP\_FIN))

            {

                tcp\_send\_control\_packet(tsk, dp->flags);

            }

            else

            {

                int pkt\_len  = dp->len + ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE + TCP\_BASE\_HDR\_SIZE;

                char \*packet = malloc(pkt\_len);

                memcpy(packet + ETHER\_HDR\_SIZE + IP\_BASE\_HDR\_SIZE + TCP\_BASE\_HDR\_SIZE, dp->packet, dp->len);

                tcp\_send\_packet(tsk, packet, pkt\_len);

            }

同时由每次重传定时器时间翻倍，更新超时时间。

p->timeout = TCP\_RETRANS\_INTERVAL\_INITIAL \* (1 << (dp->times + 1));

1. 实验结果
2. 自己对自己传输

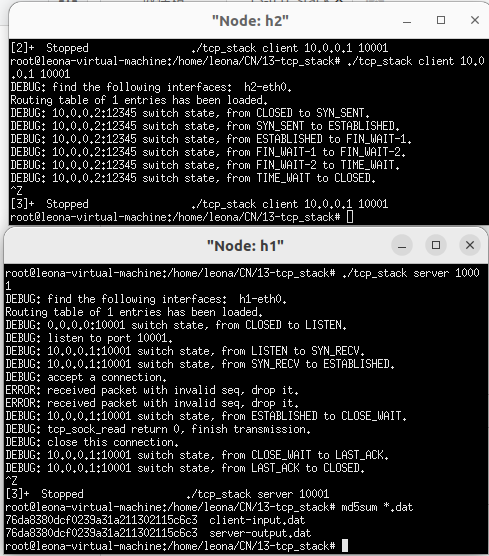


Figure ：自己对自己



Figure ：传输结果验证

1. 一段为内核版本一端为自己的代码

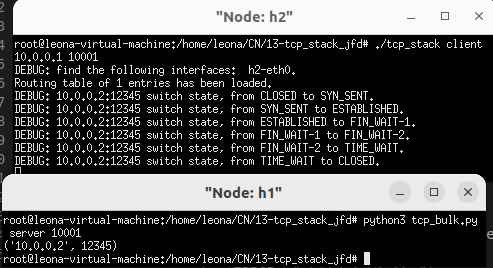


Figure ：server端为py脚本



Figure ：结果验证

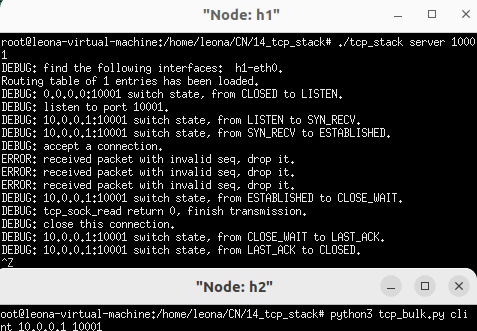


Figure ：client端为py脚本



Figure ：结果验证

1. 实验总结

本次代码量不大，主要是对于如何妥善两个队列以及计时器，在设计的时候比较耗费脑筋。本次实验让我对超时重传和TCP协议有了更深的理解。