⽹络传输机制实验二设计报告

中国科学院大学

页

2022年6月12日

一、可靠数据传输

# 实验内容

在上个实验实现了⽆丢包⽹络环境下的字符串和⽂件的传输。本次实验我们需要在上次实验的基础上实现基于超时重传的 TCP 可靠数据传输，使得节点之间在有丢包网络中能够建立连接并正确传输数据。本次实验需要完成以下内容：

①有丢包场景下的连接建立和断开；

②实现超时重传机制；

③维护发送队列。保存所有未收到 ACK的包到send\_buffer，超时时重传保存的数据；

④维护两个接收队列。有序接收队列rcv\_buf⽤来保存数据供 app 读取，⽆序接收队列rcv\_ofo\_buf⽤来保存收到的乱序的数据，数据包连续后放⼊有序接收队列；

⑤实现超时定时器。

# 实验流程

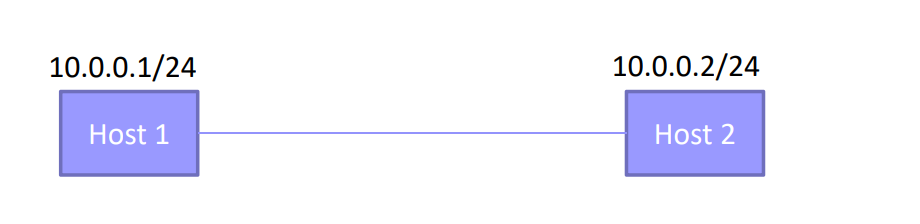


图1

①完成代码编写后编译；

②在如图 1 所示拓扑的节点 h1 上运行 TCP server，在 h2 上运行 TCP client，向 h1 发送 连接请求和关闭连接请求；在丢包率为 2%的链路上 向 h1 发送一个大小为3MB 的文件，比较 h1 接收的文件与 h2 发送的文件是否相同

③在节点 h1 上运行 TCP server，在 h2 上运行 TCP client，在丢包率为 2%的链路上 向 h1 发送一个大小为几 MB 的文件，比较 h1 接收的文件与 h2 发送的文件是否相同。然后， 在一端用tcp\_stack.py 替换 tcp\_stack 执行，用diff命令和md5sum命令验证文件的一致性。测试另一端，验证是否能和 linux 协议栈正常建立连接并且能正常传输文件。

# 实验设计

TCP的连接的建立是TCP进行可靠的数据传输的基础。后续在建立连接基础上，TCP还需要保障数据的可靠的传输。TCP根据MSS等数据，把需要传输的数据分割成TCP认为最合适的大小发送。TCP在交付了其报文给网络层后，将启动一个重传定时器，如果在该TCP报文的确认报文不能及时到达，将会启动重传。当TCP数据报到达了目的后，会发送确认报文段给源主机。这个确认不是立即发送，通常会先对报文进行校验校验出包有错，丢弃报文段，不给出响应，TCP发送数据端，超时时会重发数据。因为IP数据报的达到可能是无序的，所以我们的TCP报文段也可能是无序到达的，所以需要先对这无序的多个报文段排序后，才交付给应用层。其次如果收到的TCP报文段是重复的，将丢弃该重复的报文段。TCP需要通过序列号和确认应答得到正确顺序的数据。重发超时是指在重发数据之前，等待确认应答到来的那个特定时间间隔，如果超过这个时间仍未收到应答，就会对数据进行重发，重发后若还是收不到确认应答，则再次发送，且等待确认应答时间会以2倍、4倍的指数函数延长。数据不会无限的重发，到达一定重发次数（本次实验设置为3次）若还是收不到确认应答，就会判断为异常，强制关闭连接。

（1）发送一个新的数据包时，或者SYN|FIN： 将该数据包放进发送send\_buff末尾； 如果定时器未启动，则启动定时器。

void tcp\_send\_packet(struct tcp\_sock \*tsk, char \*packet, int len)

{

……

    tcp\_send\_buf\_append(tsk, packet, len);

    tcp\_set\_retrans\_timer(tsk);

    ip\_send\_packet(packet, len);

}

void tcp\_send\_control\_packet(struct tcp\_sock \*tsk, u8 flags)

{

   ……

*if* (flags & (TCP\_SYN | TCP\_FIN))

    {

        tsk->snd\_nxt += 1;

        tcp\_send\_buf\_append(tsk, packet, TCP\_HEADER\_LEN);

        tcp\_set\_retrans\_timer(tsk);

    }

    ip\_send\_packet(packet, TCP\_HEADER\_LEN);

}

（2）每个socket 有一个链表实现的发送队列所有没有收到 ACK的数据、 FIN 、 SYN 包，均需要在发送后放⼊发送队列中，以便超时时重传数据包。定时器和收发包线程同时访问发送队列，因此需要用锁维护。

void tcp\_send\_buf\_append(struct tcp\_sock \*tsk, char \*packet, int len)

{

    tcp\_send\_buf\_entry\_t \*entry = malloc(sizeof(tcp\_send\_buf\_entry\_t));

    entry->packet = malloc(len);

    entry->len = len;

    entry->retrans\_times = 0;

    memcpy(entry->packet, packet, len);

    pthread\_mutex\_lock(&tsk->send\_buf\_lock);

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

    pthread\_mutex\_unlock(&tsk->send\_buf\_lock);

}

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)

int tcp\_send\_buf\_retrans(struct tcp\_sock \*tsk)

{

    pthread\_mutex\_lock(&tsk->send\_buf\_lock);

    tcp\_send\_buf\_entry\_t \*entry;

    int times = 1;

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

*return* 1;

    entry = list\_entry(tsk->send\_buf.next, tcp\_send\_buf\_entry\_t, list);

*if* (entry->retrans\_times > 3)

    {

        pthread\_mutex\_unlock(&tsk->send\_buf\_lock);

*return* 0;

    }

    int len = entry->len;

    times = ++entry->retrans\_times;

    char \*packet = malloc(len);

    memcpy(packet, entry->packet, len);

    struct tcphdr \*tcp = packet\_to\_tcp\_hdr(packet);

    struct iphdr \*ip = packet\_to\_ip\_hdr(packet);

    tcp->ack = htonl(tsk->rcv\_nxt);

    tcp->checksum = tcp\_checksum(ip, tcp);

    ip\_send\_packet(packet, len);

    pthread\_mutex\_unlock(&tsk->send\_buf\_lock);

*return* times;

}

当ACK确认了部分数据，重启定时器，只需要将时间重置即可。当ACK确认了所有数据/SYN/FIN，扫描发送队列，移除 seq\_end 小于当前 ACK 的项。若发送队列清空，则关闭定时器。此外可能会收到旧的包，可能是因为不必要的重传，也可能是 ACK 丢失。如果包含数据，则回复 ACK。

static void handle\_ack(struct tcp\_sock \*tsk, struct tcp\_cb \*cb)

{

*if* (cb->flags & TCP\_ACK)

*if* (less\_than\_32b(tsk->snd\_una, cb->ack))

        {

            pthread\_mutex\_lock(&retrans\_timer\_lock);

            tsk->snd\_una = cb->ack;

            tcp\_reset\_retrans\_timer(tsk);

*if* (tcp\_send\_buf\_clear(tsk, tsk->snd\_una))

            {

                tcp\_unset\_retrans\_timer(tsk);

            }

            pthread\_mutex\_unlock(&retrans\_timer\_lock);

        }

    tcp\_update\_window(tsk, cb);

*if* (tsk->state != TCP\_LISTEN && less\_than\_32b(cb->seq, cb->seq\_end))

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

}

tcp\_send\_buf\_clear需要完成的就是扫描发送队列，移除 seq\_end 小于当前 ACK 的项，并且返回send\_buff内是否还有数据。

int tcp\_send\_buf\_clear(struct tcp\_sock \*tsk, u32 ack)

{

    pthread\_mutex\_lock(&tsk->send\_buf\_lock);

    tcp\_send\_buf\_entry\_t \*entry, \*q;

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

    {

        struct tcphdr \*tcp = packet\_to\_tcp\_hdr(entry->packet);

        struct iphdr \*ip = packet\_to\_ip\_hdr(entry->packet);

        u32 len = ntohs(ip->tot\_len) - IP\_HDR\_SIZE(ip) - TCP\_HDR\_SIZE(tcp);

        u32 seq\_end = ntohl(tcp->seq) + len + ((tcp->flags & (TCP\_SYN | TCP\_FIN)) ? 1 : 0);

*if* (less\_or\_equal\_32b(seq\_end, ack))

        {

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

            free(entry->packet);

            free(entry);

        }

*else*

*break*;

    }

    int ret = list\_empty(&tsk->send\_buf);

    pthread\_mutex\_unlock(&tsk->send\_buf\_lock);

*return* ret;

}

（3）处理乱序的数据包，如果收到的数据包是有效的（位于接收窗口内），但为乱序，则放入 ofo\_buf。否则，将数据中新的部分写入 rcv\_buf，然后检查是否可以读取 ofo\_buf到recv\_buf。

static void handle\_recv\_data(struct tcp\_sock \*tsk, struct tcp\_cb \*cb)

{

    int len = (int)(cb->seq\_end - tsk->rcv\_nxt) - (cb->flags & (TCP\_SYN | TCP\_FIN) ? 1 : 0);

*if* (!cb->pl\_len || len <= 0)

    {

        update\_tsk(tsk, cb);

*return*;

    }

    pthread\_mutex\_lock(&tsk->wait\_recv->lock);

*if* (ring\_buffer\_free(tsk->rcv\_buf) < cb->pl\_len)

        log(DEBUG, "recv buffer full.");

*else*

    {

*if* (less\_or\_equal\_32b(cb->seq, tsk->rcv\_nxt))

        {

            write\_ring\_buffer(tsk->rcv\_buf, cb->payload + cb->pl\_len - len, len);

            update\_tsk(tsk, cb);

            tcp\_check\_ofo\_buf(tsk);

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

            tsk->rcv\_wnd = max(ring\_buffer\_free(tsk->rcv\_buf), 1);

        }

*else*

            tcp\_ofo\_buf\_append(tsk, cb);

    }

    pthread\_mutex\_unlock(&tsk->wait\_recv->lock);

}

如果在 ofo队列中的乱序数据包可以组成有序数据包，则放⼊有序接收队列中。，从 seq\_end>rcv\_nxt 开始，一直读取到 seq>rcv\_nxt 时停止，检查顺序是否正确，首先检查有没有已经确认的数据包，如果有则丢弃，然后寻找当前没有确认的有序数据包，有序时放入recv\_buff。

void tcp\_check\_ofo\_buf(struct tcp\_sock \*tsk)

{

    tcp\_ofo\_buf\_entry\_t \*entry, \*q;

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

    {

*if* (less\_or\_equal\_32b(entry->seq + entry->len, tsk->rcv\_nxt))

        {

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

            free(entry->data);

            free(entry);

        }

*else* *if* (greater\_or\_equal\_32b(tsk->rcv\_nxt, entry->seq))

        {

            int len = (int)(entry->seq + entry->len - tsk->rcv\_nxt);

            write\_ring\_buffer(tsk->rcv\_buf, entry->data + entry->len - len, len);

            tsk->rcv\_nxt += len;

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

            free(entry->data);

            free(entry);

        }

*else*

*break*;

    }

}

(4)超时重传定时器

仿照之前设计的 timer\_list，设计超时重传计时器 retrans\_timer\_lis

void tcp\_set\_retrans\_timer(struct tcp\_sock \*tsk);

对timer进⾏初始化操作，并添加到链表retrans\_timer\_list中，启动重传计时器

启动定时器时，首先检查定时器的 enable 域判断是否已经启动，如果已经启动，则无须其它操作。如果定时器是关闭的，则开启并设置时间为200ms，实际本次实验的拓扑简单且传输距离不需要这么长时间可以设置5ms左右就可以，能大大加快运行时候的速度。定时器链表可能被多个线程操作，收发线程进行关闭操作，定时器线程进行扫描操作，避免发生冲突在访问时需要加锁。

void tcp\_set\_retrans\_timer(struct tcp\_sock \*tsk)

{

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

    {

        tsk->retrans\_timer.enable = 1;

        tsk->retrans\_timer.type = 0;

        pthread\_mutex\_lock(&retrans\_timer\_lock);

        list\_add\_tail(&tsk->retrans\_timer.list, &retrans\_timer\_list);

        pthread\_mutex\_unlock(&retrans\_timer\_lock);

        tsk->retrans\_timer.timeout = TCP\_RETRANS\_INTERVAL;

    }

}

void tcp\_update\_retrans\_timer(struct tcp\_sock \*tsk);

恢复计时器到最新状态重新计时。

void tcp\_unset\_retrans\_timer(struct tcp\_sock \*tsk);

从retrans\_timer\_list中删除计时器并释放空间，关闭计时器。

void tcp\_scan\_retrans\_timer\_list();

周期性扫描retrans\_timer\_list，检查是否有需要超时重传的数据包。

void \*tcp\_retrans\_timer\_thread(void \*arg);

重传计时器线程，每距离5ms到10ms调⽤⼀次tcp\_scan\_retrans\_timer\_list函数，这里本可以不用添加一个新的线程扫描retrans\_timer\_list,，直接添加到tcp\_timer\_thread中即可，但实际测试的时候这样做很浪费时间，timer\_list只有断开连接的时候才有效，需要很久一次，而丢包却时常发生，需要经常检查，因此分开后不仅可以提升效率，同时便便于找到合适的扫描间隔。

# 实验结果

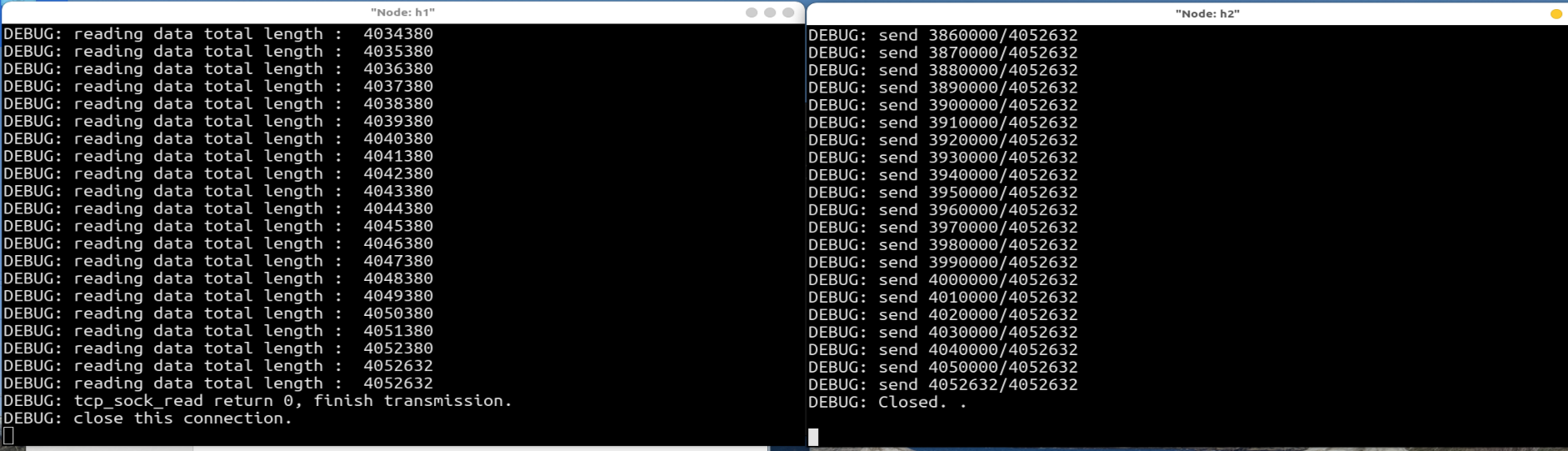


图2

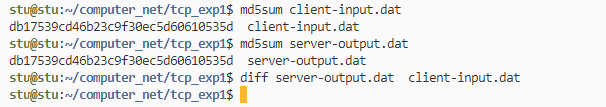


图3

TCP的可靠数据传输服务确保一个进程从其接收缓存中读出的数据流是无损坏、无间隙、非冗余和按序的数据流，即该字节流与连接的另一方端系统发出的字节流是完全相同的

从diff和md5sum的结果看一看到文件传输成功，且数据保持一致。同时通过在python中设置进程休眠时间以及丢包率，可以得到在丢包率为2%的情况下传输3MB完成需要7s左右，在丢包率为5%的情况下传输3MB完成需要11s左右

二、TCP拥塞控制机制

# 实验内容

网络中的路由器会有一个数据包处理队列，当路由器接收到的数据包太多而一下子处理不过来时，就会导致数据包处理队列过长。此时，路由器就会无条件的丢弃新接收到的数据封包。这就会导致上层的 TCP 协议以为数据包在网络中丢失，进而重传这些数据包，而路由器又会丢弃这些重传的数据包，如此以往，就会导致网络性能急剧下降，引起网络瘫痪。因此，TCP 需要控制数据包发送的数量来避免网络性能的下降。

本次实验需要在实现协议栈的可靠传输的基础之上，了解 TCP拥塞控制机制，理解数据包发送的条件实现维护拥塞控制状态迁移、根据当前状态控制数据包发送以及实现快重传、快恢复机制。最后使用有丢包网络拓扑脚本进行测试，验证该实现的正确性，并记录 中每次 cwnd 调整的时间和相应值，呈现到二维坐标图中

# 实验流程

1. 添加拥塞控制阶段代码以及保存cwnd大小变化线程代码并编译；
2. 执行tcp\_stack\_loss.py在节点h1上执行TCP程序在h1上运行TCP协议栈的服务器模式 (./tcp\_stack server 10001)在h2上运行TCP协议栈的客户端模式 (./tcp\_stack client 10.0.0.1 10001)

    h1.cmd('./tcp\_stack server 10001 > h1-output.txt 2>&1 &')

    h2.cmd('./tcp\_stack client 10.0.0.1 10001 > h2-output.txt 2>&1 &')

    sleep(30)

*for* r *in* (h1, h2):

        r.cmd('pkill -SIGTERM tcp\_stack')

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

使用md5sum和diff比较两个文件是否完全相同；

1. 记录传输中每次cwnd调整的时间和相应值，呈现到二维坐标图中。

# 实验设计

（1）拥塞状态迁移

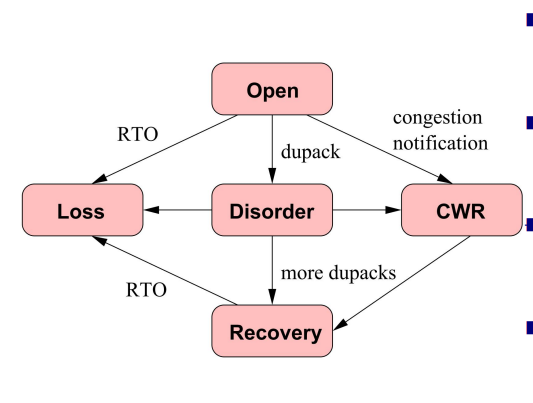


图4

Open：当没有丢包且没有收到重复的ACK时，处于该状态，同时也是初始状态。若此时收到了新的ACK，则增加拥塞窗口值，若收到了重复的ACK，则状态转移到Disorder。

Disorder：收到了重复的ACK，但重复数量还不足以触发重传，则继续处于Disorder状态。若收到了三个重复的ACK，判定丢包，启动快重传，转移到下一个状态Recovery。同时，更新ssthresh的值，标记拥塞窗口大小应减小。

Recovery:遇到网络丢包时，处于该状态。此时，进入快重传快恢复阶段，先将窗口值减半，然后恢复丢包当所有的包都已确认接收，回到 OPEN 状态；否则重传一个包。如果 ACK 包是无效的 dupacks 加一，在途数据包减少了一个，因此唤醒发送进程，可以再发送一个数据包。如果触发了超时重传，那么进入Loss状态.

Loss:当触发超时重传时，处于该状态。进入时阈值减为 cwnd 的一半，cwnd 减为 1，重新开始慢启动。LOSS 状态确认丢失的包全部接收后，转为 OPEN 状态。

*switch* (tsk->cs\_state) {

*case* OPEN: {

*if* (ack\_valid) {

            update\_cwnd(tsk);

*}else*{

                tsk->dupacks ++;

                tsk->cs\_state = DISORDER;

            }

*break*;

        }

*case* DISORDER: {

*if* (tsk->cwnd < tsk->ssthresh)

                tsk->cwnd += 1;

*else*

                tsk->cwnd += 1.0 / tsk->cwnd;

*if* (!ack\_valid) {

                tsk->dupacks ++;

*if* (tsk->dupacks >= 3) {

                    tsk->ssthresh = max((u32)(tsk->cwnd / 2), 1);

                    tsk->cwnd -= 0.5;

                    tsk->cwnd\_flag = 0;

                    tsk->recovery\_point = tsk->snd\_nxt;

                    tsk->cs\_state = RECOVERY;

                    tcp\_retrans\_send\_buffer(tsk);

                }

            }

*break*;

        }

*case* LOSS: {

*if* (tsk->cwnd < tsk->ssthresh)

                tsk->cwnd += 1;

*else*

                tsk->cwnd += 1.0 / tsk->cwnd;

*if* (ack\_valid) {

*if* (cb->ack >= tsk->loss\_point) {

                    tsk->cs\_state = OPEN;

                    tsk->dupacks = 0;

                }

            }

*else* {

                tsk->dupacks ++;

            }

*break*;

        }

*case* RECOVERY: {

*if* (tsk->cwnd > tsk->ssthresh && tsk->cwnd\_flag == 0)

                tsk->cwnd -= 0.5;

*else*

                tsk->cwnd\_flag = 1;

*if* (ack\_valid) {

*if* (cb->ack < tsk->recovery\_point) {

                    tcp\_retrans\_send\_buffer(tsk);

                }

*else* {

                    tsk->cs\_state = OPEN;

                    tsk->dupacks = 0;

                }

            }

*else* {

                tsk->dupacks ++;

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

            }

*break*;

        }

*default*:

*break*;

    }

（2）拥塞避免算法

当主机开始发送数据时，如果立即将大量数据字节注入到网络，那么就有可能因为不清楚当前网络的负荷情况而引起网络阻塞。所以，最好的方法是先探测一下，即由小到大逐渐增大发送窗口，也就是说，由小到大逐渐增大拥塞窗口数值。通常在刚刚发送报文段时，先把拥塞窗口cwnd设置为一个最大报文段MSS的数值。而在每收到一个新的报文段的确认后，把拥塞窗口增加至多一个MSS的数值。用这样的方法逐步增大发送方的拥塞窗口cwnd，可以使分组注入到网络的速率更加合理。（慢开始当中的“慢”并不是指cwnd的增长速率慢，而是在TCP开始发送报文段时先设置cwnd = 1,使得发送方在开始时只发送一一个报文段）当rwnd足够大的时候，为了防止拥塞窗口cwnd的增长引起网络阻塞，还需要另外一个变量------慢开始门限ssthresh。

    当cwnd < ssthresh时，使用上述慢启动算法；而当cwnd >ssthresh时，停止使用慢启动算法，改用拥塞避免算法；

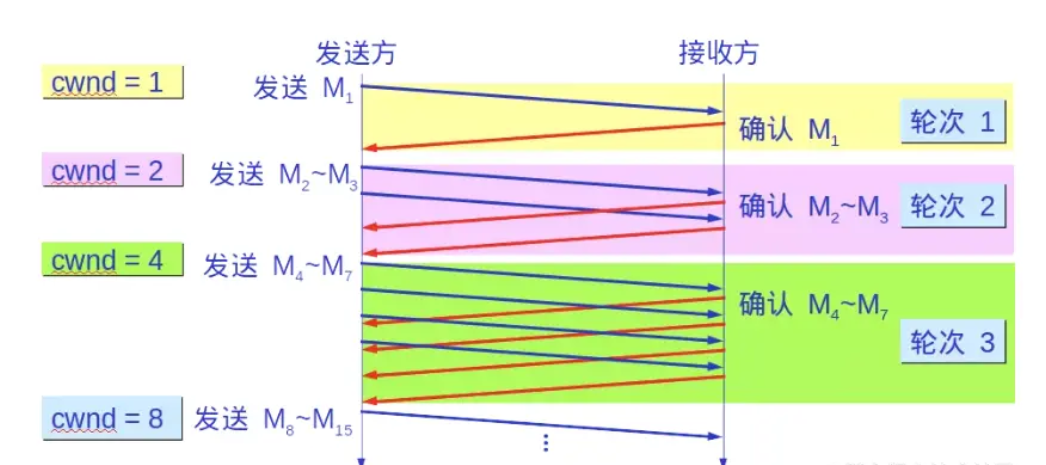
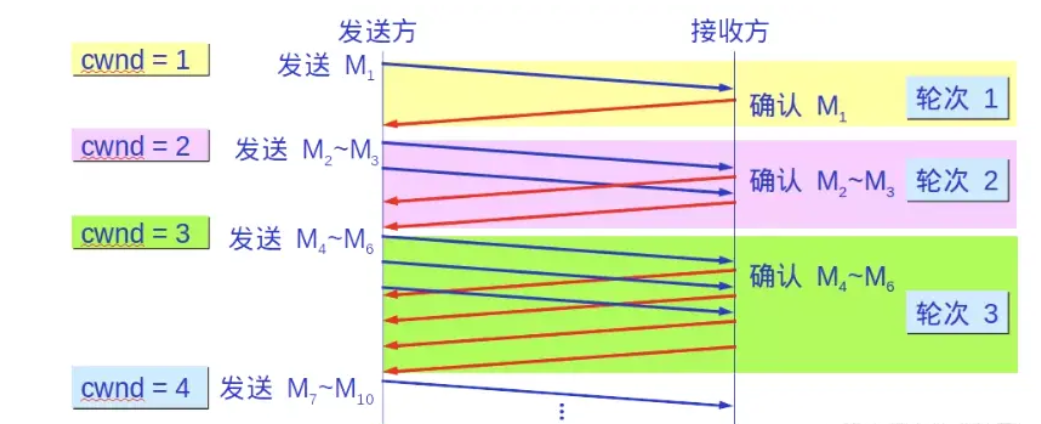


图5

慢启动算法有一定的局限性。首先需要获得网络内部流量分布的信息，浪费可用的网络容量，额外开销；其次估算合理的ssthresh值并不容易，可能耗时较长；

拥塞避免让拥塞窗口cwnd缓慢的增大，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍，这样拥塞窗口cwnd按线性规律缓慢的增长，比慢开始算法的拥塞窗口增长速率缓慢的多。无论是慢启动算法还是拥塞避免算法，只要判断网络出现拥塞，就要把慢启动开始门限（ssthresh）设置为发送窗口的一半（>=2），cwnd设置为1，然后再使用慢启动算法，这样做的目的能迅速的减少网络当中的数据传输，使发生拥塞的路由器能够把队列中堆积的数据处理完毕。



在拥塞控制中，慢启动算法和拥塞避免算法 是怎么配合使用的呢？

像上面所说，慢启动算法下的cwnd大小是指数增长，所以不能任 cwnd 任意增长，所以我们引入一个慢启动门限（ssthresh）的阈值来控制 cwnd 的增长。

ssthresh的作用是：

当cwnd < =ssthresh时，使用慢开始算法。

当cwnd > ssthresh时，改用拥塞避免算法。

还有一个问题就是这个 ssthresh 是怎么设置的呢？

TCP/IP 中规定无论是在慢开始阶段还是在拥塞避免阶段，只要发现网络中出现拥塞（没有按时收到确认），就要把ssthresh设置为此时发送窗口的一半大小（不能小于2）。

TCP 的可靠传输的原理就是超时重传机制，而重发机制有两种：超时重传机 和 快重传。TCP超时重传机制，结合上面说的慢启动和拥塞避免使用就是发送完数据后开始倒计时，如果在重发超时内没有收到对接收方发来的ACK的话就去执行上述的乘法减小过程（设置sstresh为原来一半） 并重新开始慢开始算法，重新发送数据。TCP快重传， 则是允许发送方再连续收到3 个重复的确认后就可以开始执行乘法减小过程而不必再等待重发超时时间。这就需要接收方每收到一个失序的报文段就立即发出重复确认以让发送发及早知道有报文段丢失，而不是等待自己发送数据的时候进行捎带确认。

快恢复算法是与快重传算法配合使用的一个算法。快恢复主要是指，当快重传的时候，发送方快速收到了3个重复的确认，因此会认为网络不是拥塞状态，所以在乘法减小过程（设置sstresh为原来一半），会启动 “拥塞避免”，而不是TCP超时重发机制的重新启动的慢启动

（3）保存拥塞窗⼝⼤⼩并画图在 tcp\_timer.c 中增加函数 tcp\_cwnd\_thread ，⽤来新建⼀个保存 cwnd 的线程

void \*tcp\_cwnd\_thread(void \*arg)

{

    struct tcp\_sock \*tsk = (struct tcp\_sock \*)arg;

    FILE \*fp = fopen("cwnd.txt", "w");

    int time = 0;

*while* (tsk->state == TCP\_ESTABLISHED)

    {

        usleep(50);

time += 500;

        fprintf(fp, "%d %f\n", time, tsk->cwnd);

    }

    fclose(fp);

*return* NULL;

}

# 实验结果

Md5sum和diff验证

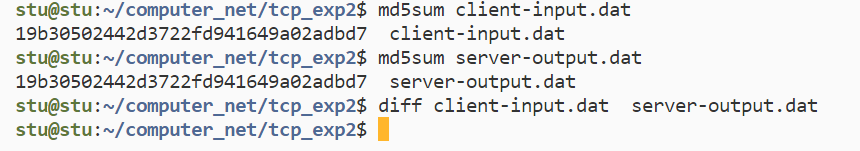
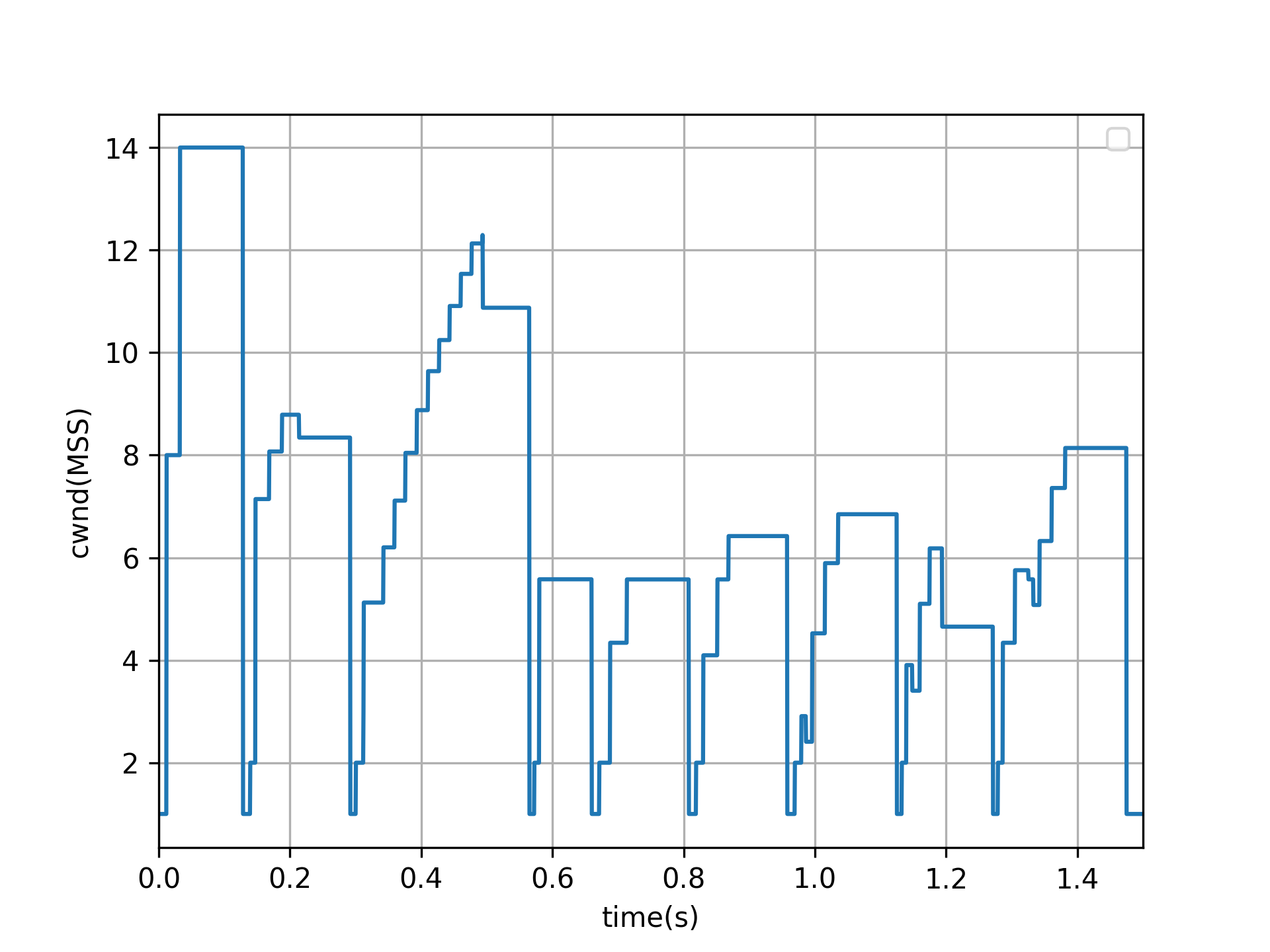


图5

文件保持一致，同时从重传次数上看丢包重传的次数与之前相⽐有下降，但从实际的完成需要时间看，加上拥塞控制之后实际传输速度满了许多。这可能是本次拓扑线路比较简单，没有很好的发挥拥塞控制的优势，在复杂拓扑下拥塞控制理论上能很好的缓解网络瘫痪，提高网络性能。



曲线走势和理论大致相同，基本符合预期。

# 实验总结

TCP协议栈通过流量控制控制发送者的发送速度从而使接收者来得及接收，防止丢包。而拥塞控制则是防止过多的数据注入到网络中，避免网络中的路由器或链路不致过载，导致数据丢失，从而保证了TCP传输的可靠性。 加上上一个实验 TCP协议可靠性是如何保证之滑动窗口，超时重发，序列号确认应答信号，那么关于TCP可靠性的机制就实现了。