网络传输机制实验四报告

李昊宸

2017K8009929044

（一）TCP拥塞控制实现

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

5）记录h2中每次cwnd调整的时间和相应值，呈现到二维坐标图中

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

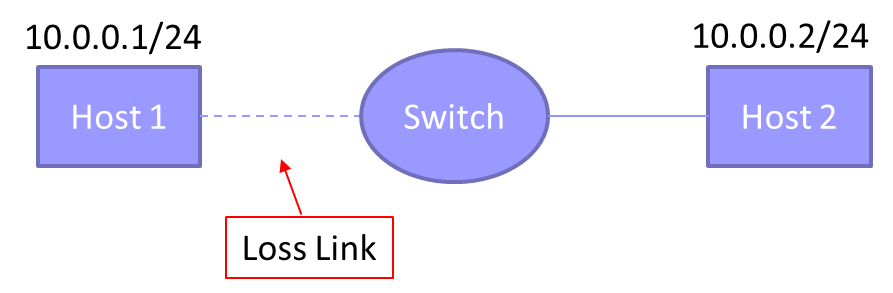
tcp\_stack.py ： python应用实现，用于测试

tcp\_timer.c ： TCP定时器

create\_randfile.sh # 随机生成文件的脚本

cwnd\_result.py : 将提取的cwnd-时间数据进行画图

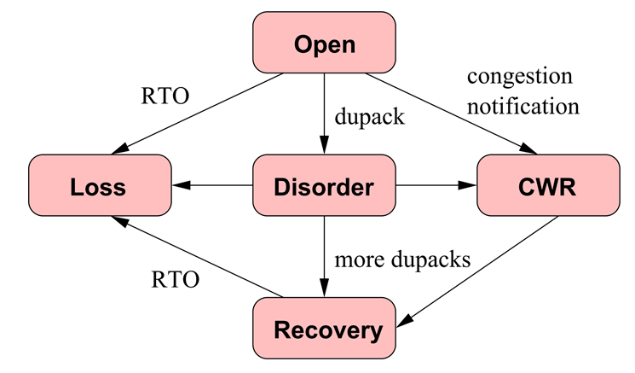
tcp\_topo\_loss.py： 实现二节点的丢包率为2%的拓扑



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的可靠传输，这周需要补充可靠传输期间的拥塞控制。

TCP拥塞控制机制：



图二 TCP拥塞控制状态转移图

Open：没有出现丢包/没有收到重复的ACK

机制：慢启动/拥塞避免，收到ACK后增加拥塞窗口值

Disorder：收到了重复的ACK，但是还不够触发快重传的量

机制：慢启动/拥塞避免，收到ACK后增加拥塞窗口值

Recovery：重复ACK达到重传所需量，快恢复重传ACK对应的数据包

机制：窗口值减半，ssthresh设为窗口值，拥塞避免

Loss：触发超时重传定时器

机制：ssthresh设为窗口值的一半，窗口值设为1，慢启动

拥塞窗口增大：

1）慢启动：

对方每确认一个报文段，cwnd增加1MSS，直到cwnd超过ssthresh值。

经过1个RTT，前一个cwnd的所有数据被确认后， cwnd大小翻倍。

2）拥塞避免：

对方每确认一个报文段，cwnd增加 ∗ 1MSS

经过1个RTT，前一个cwnd的所有数据被确认后， cwnd增加1 MSS

拥塞窗口减小：

1）快重传：

Ssthresh减小为当前cwnd的一半：ssthresh <- cwnd / 2

新拥塞窗口值cwnd <- 新的ssthresh

2）超时重传：

Ssthresh减小为当前cwnd的一半：ssthresh <- cwnd / 2

拥塞窗口值cwnd减为1 MSS

拥塞窗口不变：

快恢复：

1）进入：快重传触发之后立即进入

2）退出：一方面，对方确认了进入快恢复前发送的所有数据时，进入Open状态

另一方面，如果触发超时重传，就进入Loss状态

3）期间收到ACK：如果该ACK没有确认新数据，说明inflight数据包数目减少一

个，cwnd允许发送一个新数据包。

如果该ACK确认了新数据：如果是部分确认，则重传对应的数据

包；否则退出快恢复阶段，进入Open状态。

数据包重传：

判断丢包发生：1）Open状态下收到重复的3个ACK

2）快恢复状态下收到部分确认ACK

3）超时重传计时器过期

恢复丢包：1）快重传：1个RTT

2）快恢复：1-n个RTT（n为重传后丢包的个数）

3）超时重传：1个RTO

拥塞控制下的数据包发送：当网络中在途数据包的数目小于发送窗口大小时，允许发送数据包。

snd\_wnd = min(adv\_wnd, cwnd)

发送窗口等于接收窗口和拥塞窗口中的最小值

inflight = (snd\_nxt - snd\_una)/1MSS - #(dupacks) - #(loss) + #(retrans)

在途数据包等于发送包序号减确认包序号，再减重复ACK个数，减丢包个数，再加重传

包的个数。

#(packets allowed to send) = max(snd\_wnd / 1MSS - inflight, 0)

发送窗口减在途数据包如果大于0，就可以继续发送。

较为缓和的窗口变化机制：

1）窗口大小减半：

如果cwnd立即减半，cwnd会小于inflight，一段时间内不能发送任何包，可能出

现死锁bug。可以设置每收到一个ACK，就将cwnd减0.5，这样在一个RTT内窗口能

大致减半，使发送不会突然截止。

2）拥塞避免阶段窗口增加：

每收到一个ACK，就将cwnd增加1/cwnd，这样在一个RTT内窗口能大致增加1

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

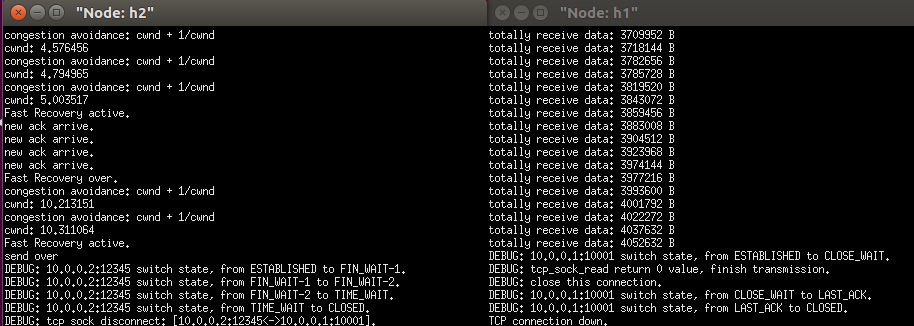
h2# python tcp\_stack.py client 10.0.0.1 10001

mininet> quit

sudo python cwnd\_result.py

三、实验结果及分析

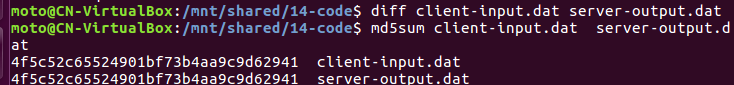
1. 实验结果



图三 TCP可靠传输实验结果

可以看到，每收到一个最新ACK，拥塞避免下cwnd会增长1/cwnd。当出现3个连续重复ACK后，会触发快重传和快恢复。快重传和快恢复后，继续执行拥塞避免。

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。



图四 TCP可靠传输文件比较实验结果

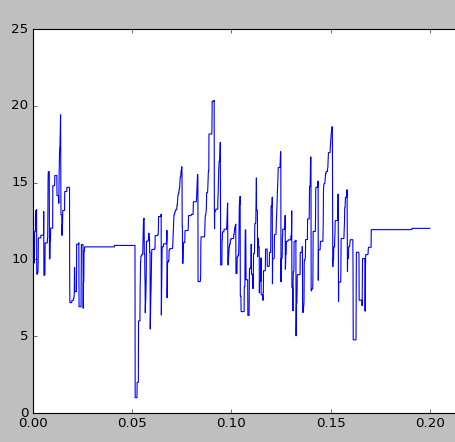
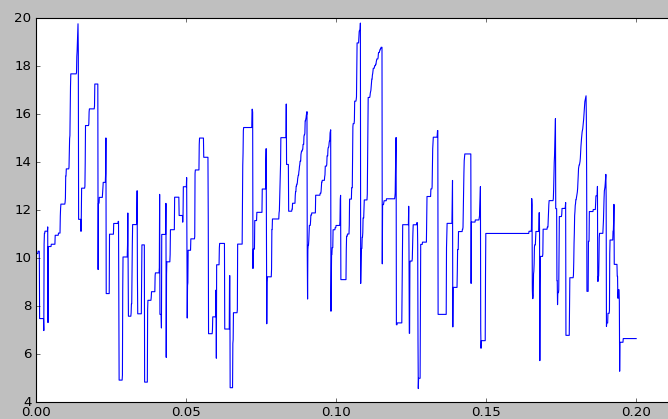
调用diff命令比较client\_input.dat和server-output.dat，发现完全一致。

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

说明新机制的加入，不影响稳定传输。

需要说明的是，由于快重传和快恢复机制的加入，整个过程中未发生过任何一次超时重

传。整个数据流的传输速度提高到了原先的5倍左右。



图五 不同时隙下的cwnd-time二维曲线

采取不同的发送时隙，记录cwnd-time数值并进行绘图，得到以上两个类似的曲线。图中可以看到慢启动的指数上升，拥塞避免的线性上升，以及触发重传后的指数下降至原先的一半后继续线性增长。复现效果基本成功。

附：遵从Linux协议栈实现，cwnd的单位为数据包个数。

（二）实验代码详解

本次实验代码过长，不再赘述代码的内容。。

1. tcp\_in.c:修改tcp\_process的处理逻辑

（1）void tcp\_process(struct tcp\_sock \*tsk, struct tcp\_cb \*cb, char \*packet)：

快重传和快恢复主要用于ESTABLISHED阶段的通信，于是主要修改ESTABLISHED阶段收到ACK后的处理逻辑。

如果当前状态是ESTABLISHED，并且收到的报文是ACK报文，而且不是FIN，就进行以下修改：不再调用send\_buffer\_ACK，而是进行手动确认：

1.遍历发送缓冲区，删除所有seq小于ACK的数据包缓存

2.如果有缓存被删除，说明收到的确认是新的确认号，即有新的有序数据包被确认：

1）如果当前状态为OPEN、DISORDER、RECOVERY、LOSS：

如果cwnd小于ssthresh，就执行慢启动：cwnd++

如果cwnd大于等于ssthresh，就执行拥塞避免：cwnd+=1/cwnd。

如果当前状态不为LOSS，或者为LOSS但是ACK号大于losspoint（意味

着超时重传发生的瞬间，重传了所有的发送缓存，现在都已经被确认）

状态就转移到OPEN

2）如果当前状态为FR（Fast Recovery）：

如果cwnd此时处于上升阶段（fr\_flag为1），就就执行拥塞避免：

cwnd+=1/cwnd。

否则，如果cwnd大于ssthresh（处于下降减半阶段），就cwnd-=0.5。

否则，（处于下降减半结束，准备继续增长）设置fr\_flag=1。

之后，如果ACK小于recovery\_point（意味着快重传发生的瞬间，重传

了发送缓存，现在还有未被确认的），就调用

send\_buffer\_RETRAN\_HEAD重传发送缓存中的第一个数据包。

否则（意味着快恢复完成），状态转移到OPEN

3.如果没有缓存被删除，说明收到的确认号还是老确认号：

1）如果当前状态为OPEN、DISORDER、LOSS：

如果cwnd小于ssthresh，就执行慢启动：cwnd++

如果cwnd大于等于ssthresh，就执行拥塞避免：cwnd+=1/cwnd。

如果当前状态为OPEN，就转移到DISORDER。（第二个重复ACK）

如果当前状态为DISORDER，就转移到RECOVERY。（第三个重复ACK）

2）如果当前状态为RECOVERY：

ssthresh设置为当前cwnd的一半（至少为1）。

cwnd-=0.5。

fr\_flag=0（cwnd开始下降）。

设置recovery\_point为snd\_nxt，表示发送缓存中所有数据都需要恢复。

调用 send\_buffer\_RETRAN\_HEAD重传发送缓存中的第一个数据包。

设置状态转移为FR（Fast Recovery）

3）如果当前状态为FR：

如果cwnd此时处于上升阶段（fr\_flag为1），就就执行拥塞避免：

cwnd+=1/cwnd。

否则，如果cwnd大于ssthresh（处于下降减半阶段），就cwnd-=0.5。

否则，（处于下降减半结束，准备继续增长）设置fr\_flag=1。

1. tcp\_timer.c:增加cwnd记录

（1）void \*tcp\_cwnd\_plot\_thread(void \*arg)：每50微秒记录一次当前cwnd的值

没啥好说的，直接上代码

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

FILE \*file = fopen("cwnd.dat", "w");

float i = 0;

while (tsk->state != TCP\_TIME\_WAIT)

{

usleep(50);

++i;

fprintf(file, "%f:%f\n",i/10000, tsk->cwnd);

}

fclose(file);

return NULL;

在进入ESTABLISHED时，create\_pthread新建一个线程，用于运行该程序。