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

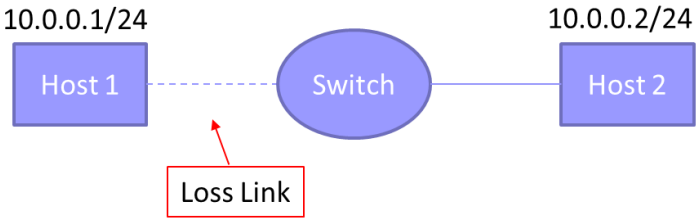
5）记录 h2 中每次 cwnd 调整的时间和相应值，呈现到二维坐标图中

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
```

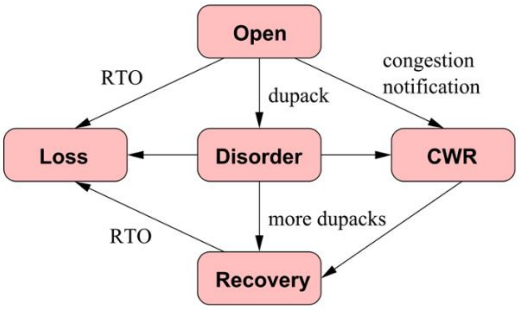
cwnd_result.py ： 将提取的 cwnd-时间数据进行画图
tcp_topo_loss.py： 实现二节点的丢包率为 2%的拓扑



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$\frac{1\,MSS}{CWND}$ * 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

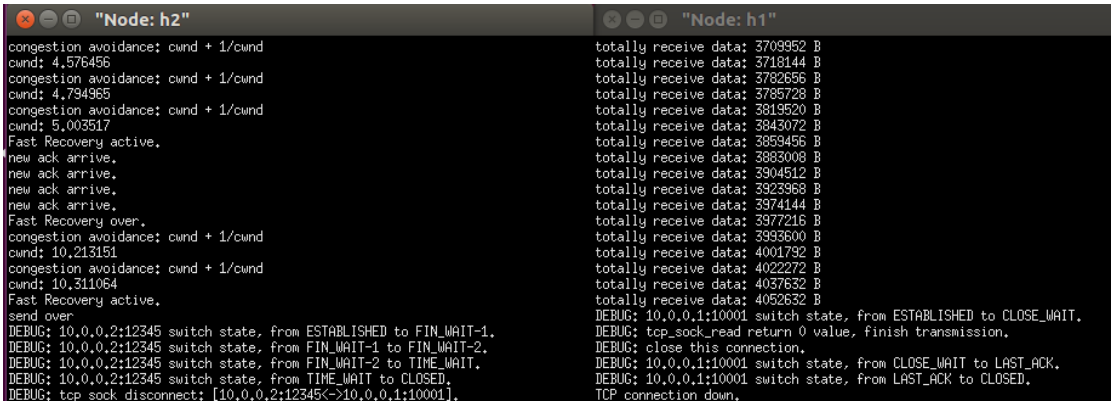
2. 启动脚本

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

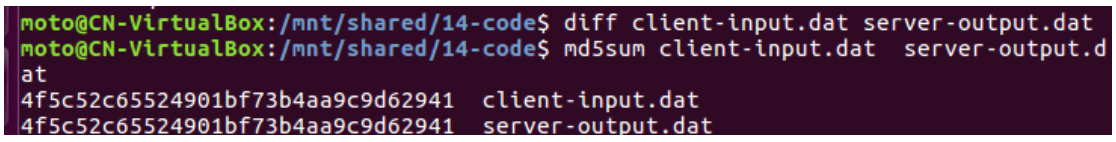
1. 实验结果



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。

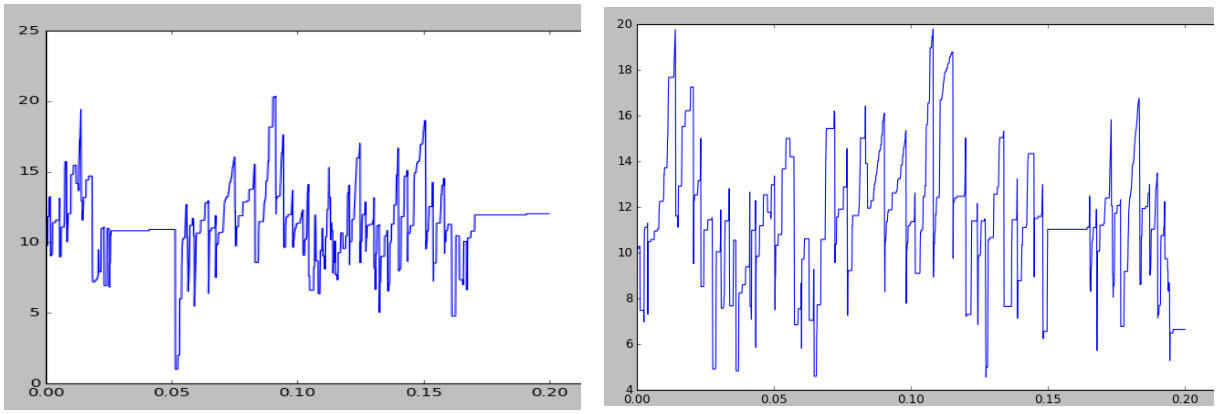
在数据传输完成后（达到了文件大小 4052632B），client（h2）发起关闭连接的请求，server（h1）响应之。整个状态变化过程与上一次实验相同。



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。。

一、 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。

二、 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。