网络传输机制实验二

武庆华

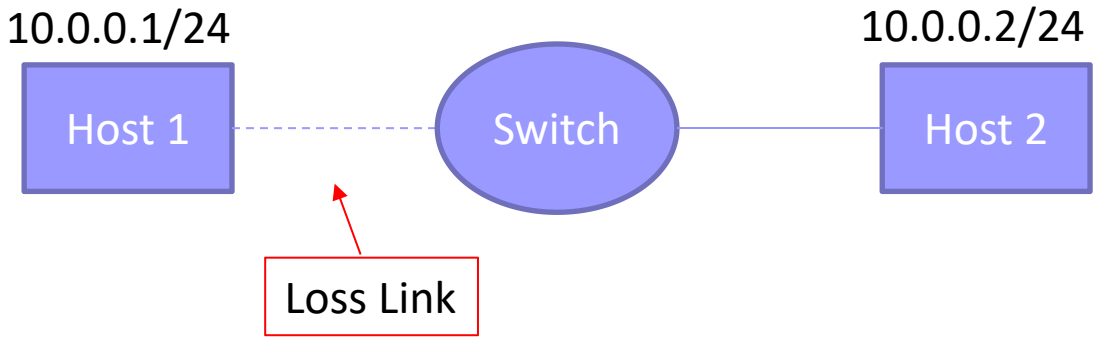
wuqinghua@ict.ac.cn

主要内容

- 可靠数据传输
  - 网络丢包
  - 超时重传机制
  - 有丢包场景下的连接建立和断开
  - 发送队列和接收队列
  - 超时定时器实现
- TCP拥塞控制机制
  - 拥塞控制状态迁移
  - 拥塞控制机制设计
  - 拥塞控制机制实现

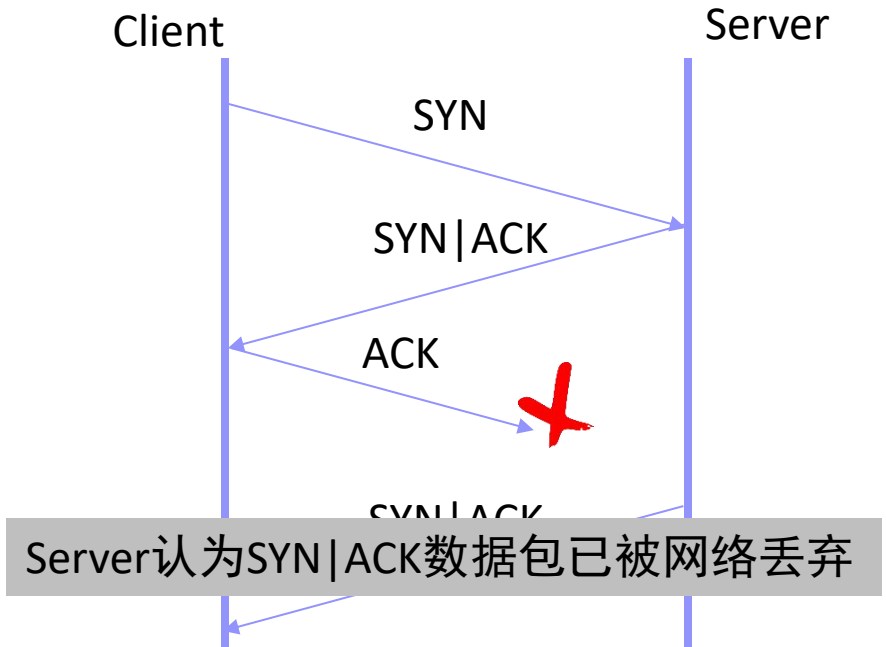
网络传输机制实验

- 实验目的：有丢包场景的可靠传输
  - 丢包恢复：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
  - 拥塞控制：实现TCP NewReno拥塞控制机制，发送方能够根据网络拥塞（丢包）信号调整拥塞窗口大小

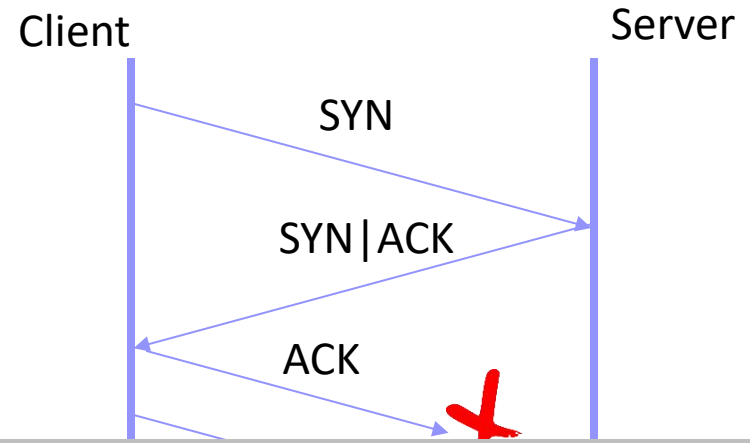


网络丢包(Packet Drop vs Packet Loss)

- 带数据/SYN/FIN的包超过一定时间没被确认
  - 带数据的包被丢弃，超过一定时间未收到对应ACK，发送方认为该包丢失
  - 带数据的包没有丢弃，但其对应ACK被丢弃，发送方会认为该包丢失
    - ACK: Packet Drop, Data: Packet Loss
  - 网络丢弃ACK数据包，也可能不被双方感知
    - ACK: Packet Drop, No Packet Loss



Server认为SYN|ACK数据包已被网络丢弃

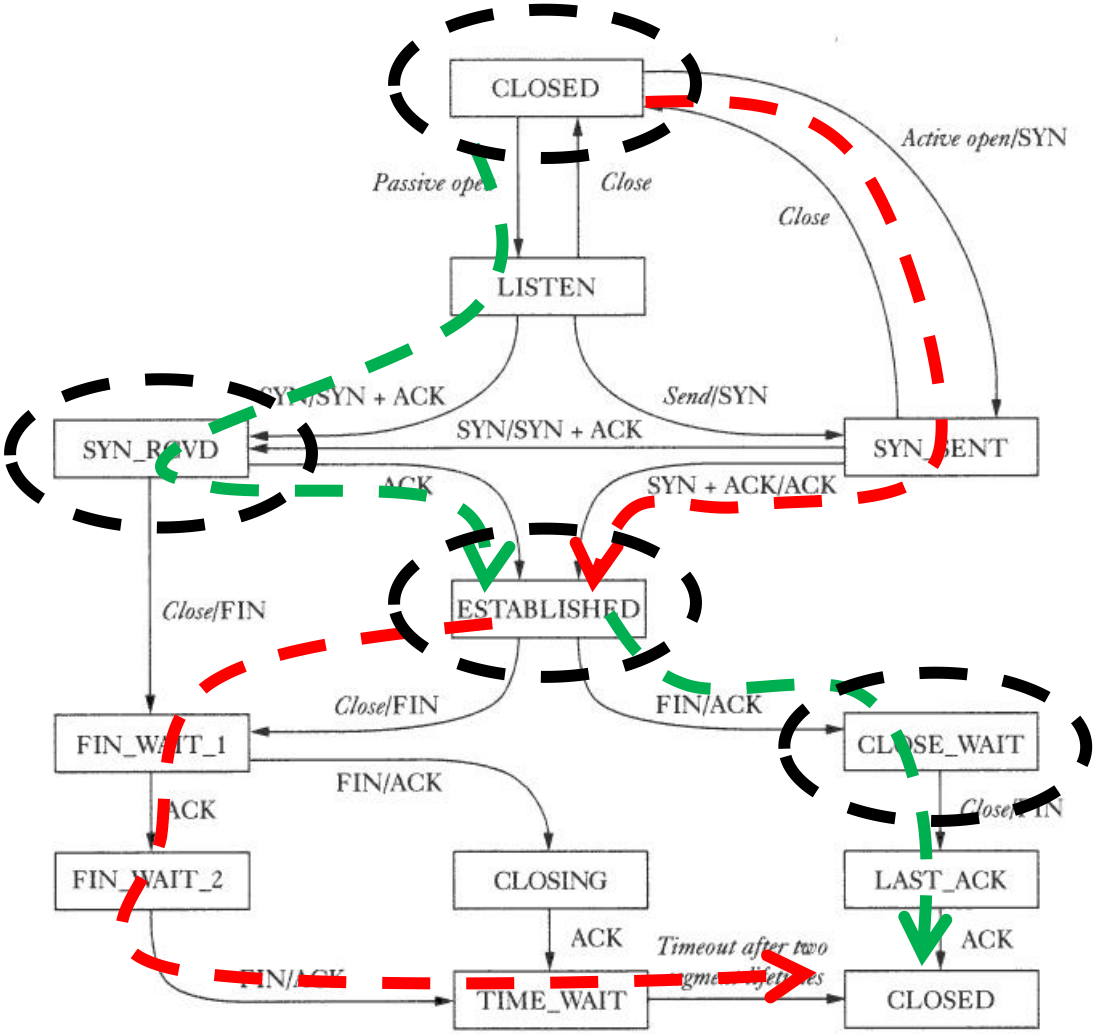


由于Data数据包中的ACK确认了Server的SYN|ACK，即使前面的ACK丢失，不需要重传也能建立起连接

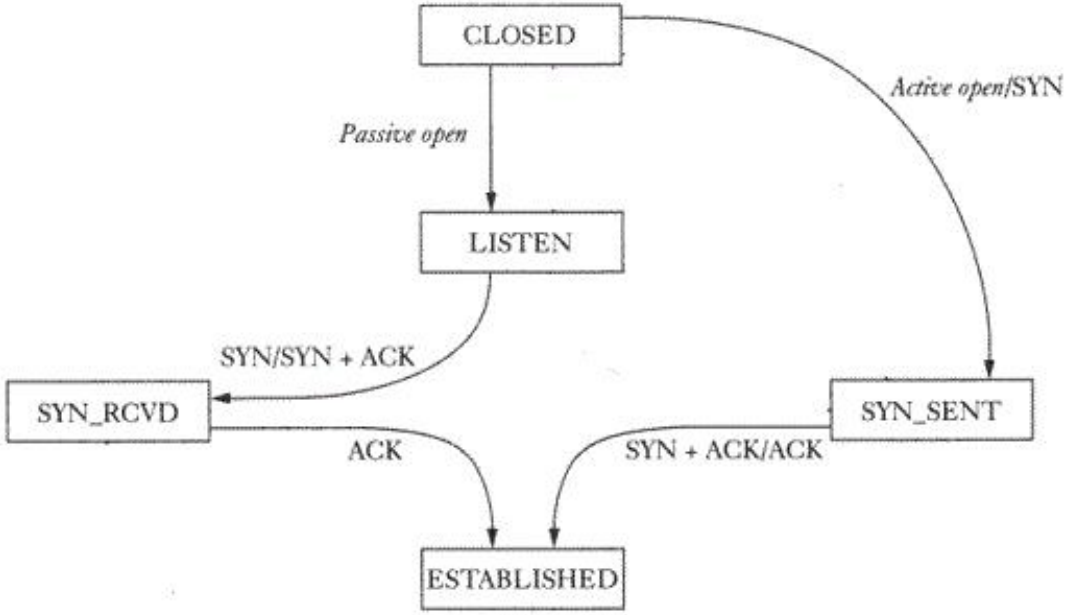
超时重传机制

- 每个连接维护一个超时重传定时器
- 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
- 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)

发生丢包的位置


CLOSED
Active open/SYN
Passive open
Close
Close
LISTEN
SYN/SYN + ACK
Send/SYN
SYN/SYN + ACK
SYN_RCVD
SYN_SENT
ACK
SYN + ACK/ACK
Close/FIN
ESTABLISHED
Close/FIN
FIN/ACK
FIN_WAIT_1
CLOSE_WAIT
FIN/ACK
ACK
FIN_WAIT_2
CLOSING
LAST_ACK
ACK
ACK
Timeout after two
TIME_WAIT
CLOSED

有丢包时的连接建立



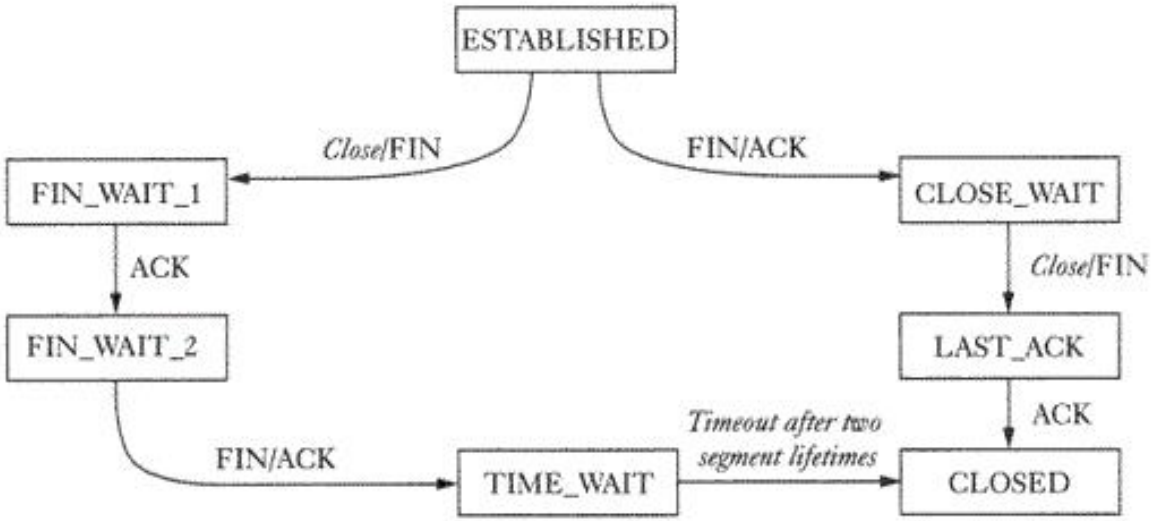
主动建立连接

- Case #1: 发送SYN，该数据包被丢弃
  - Active: SYN_SENT, Passive: LISTEN
- Case #2: 对方发送的SYN|ACK被丢弃
  - Active: SYN_SENT, Passive: SYN_RCVD

被动建立连接

- Case #3: 发送的SYN|ACK被丢弃
  - 同Case #2
- Case #4: 对方发送的ACK被丢弃
  - Active: ESTABLISHED, Passive: SYN_RCVD

有丢包时的连接断开



主动关闭连接

- Case #1: 发送的FIN_1被丢弃
  - Active: FIN_WAIT_1, Passive: ESTABLISHED
- Case #2: 对方发送的ACK_1被丢弃
  - Active: FIN_WAIT_1, Passive: CLOSE_WAIT

被动关闭连接

- Case #3: 发送的FIN_2被丢弃
  - Active: FIN_WAIT_2, Passive: LAST_ACK
- Case #4: 对方发送的ACK_2被丢弃
  - Active: TIME_WAIT, Passive: LAST_ACK

如何处理网络丢包

- 由发送数据(包括SYN和FIN)的一方负责重传
  - 连接建立和断开时的丢包只能超时重传
  - 数据传输过程中的丢包，满足快速重传条件的，可以快速重传
- 如果是数据/SYN/FIN包被丢弃
  - 发送方重传该数据包，不需要切换状态
    - 其在第一次发送时已经改变了状态
  - 接收方按正常流程处理
    - 其不能区分是否为重传数据包
- 如果是ACK包被丢弃
  - 发送方重传该数据包，不需要切换状态
  - 接收方相当于多次接收了该数据包，不切换状态，但要检查该数据包是否合法并回复ACK，包括：是否能够触发切换到该状态，seq/ack是否正确
- 如果既包含数据/SYN/FIN，又包含更新的ACK，。。。

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

- 发送新的数据时
  - 放到snd_buffer队尾，打开定时器
- 收到新的ACK
  - 将snd_buffer中seq_end <= ack的数据包移除，并更新定时器
- 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

接收队列

- 数据接收方需要维护两个队列
  - 已经连续收到的数据，放在rcv_ring_buffer中供app读取
  - 收到不连续的数据，放到rcv_ofo_buffer队列（链表实现）中
- TCP属于发送方驱动传输机制
  - 接收方只负责在收到数据包时回复相应ACK
- 收到不连续的数据包时
  - 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

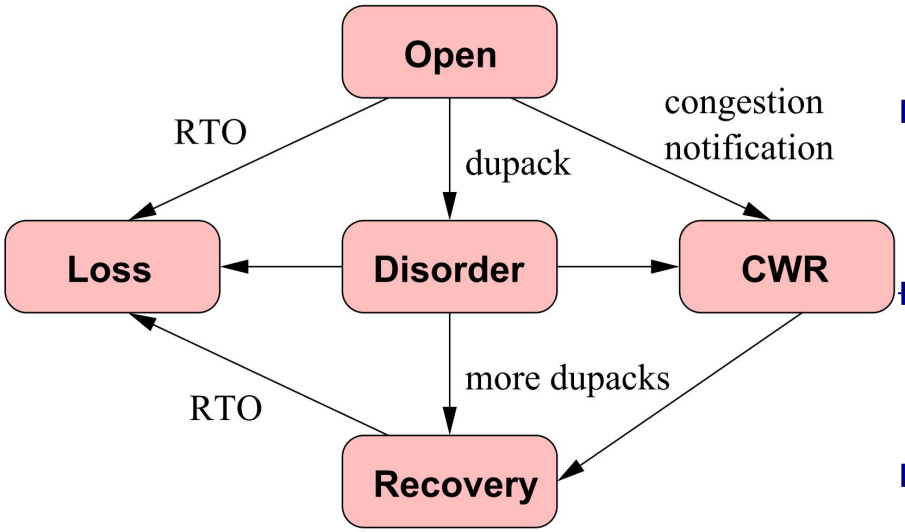
超时重传实现

- 在tcp_sock中维护定时器
  - struct tcp_timer retrans_timer;
- 当开启定时器时
  - 将retrans_timer放到timer_list中
- 关闭定时器时
  - 将retrans_timer从timer_list中移除
- 定时器扫描
  - 建议每10ms扫描一次定时器队列，重传定时器的值为200ms * 2^N

TCP实验内容一：丢包恢复

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo_loss.py)
- 在节点h1上执行TCP程序
  - 执行脚本(disable_offloading.sh , disable_tcp_rst.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 (./tcp_stack server 10001)
- 在节点h2上执行TCP程序
  - 执行脚本(disable_offloading.sh, disable_tcp_rst.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 (./tcp_stack client 10.0.0.1 10001)
    - 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比较两个文件是否完全相同
- 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

TCP拥塞控制状态迁移图



- Open: 没有丢包/重复ACK
  - 收到ACK后增加拥塞窗口值
- Disorder: 收到重复ACK，不够触发重传
  - 同Open状态
- CWR: 收到ECN通知
  - 窗口大小减半
- Recovery: 遇到网络丢包
  - 窗口值减半，恢复丢包
- Loss: 触发超时重传定时器
  - 认为所有未确认的数据都丢失
  - 窗口从1开始慢启动增长

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)
  - inflight = (snd_nxt - snd_una)/1MSS - #(dupacks) - #(loss) + #(retrans)
  - #(packets allowed to send) = max(snd_wnd / 1MSS - inflight, 0)

Loss为估计值，Retrans为实际值，理论上两者应该相等

TCP拥塞窗口增大

- 慢启动（Slow Start）
  - 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
- 拥塞避免（Congestion Avoidance）
  - 对方每确认一个报文段，cwnd增加$\frac{1\ \mathrm{MSS}}{\mathrm{CWND}} * 1\mathrm{MSS}$
  - 经过1个RTT，前一个cwnd的所有数据被确认后， cwnd增加1 MSS

```
New ack received:
    if cwnd < ssthresh: # Slow Start
        cwnd = cwnd + 1
    else: # Congestion Avoidance
        cwnd = cwnd + 1/cwnd
```

TCP拥塞窗口减小

- 快重传（Fast Retransmission）
  - Ssthresh减小为当前cwnd的一半：ssthresh <- cwnd / 2
  - 新拥塞窗口值cwnd <- 新的ssthresh
- 超时重传（Retransmission Timeout）
  - Ssthresh减小为当前cwnd的一半：ssthresh <- cwnd / 2
  - 拥塞窗口值cwnd减为1 MSS

TCP拥塞窗口不变

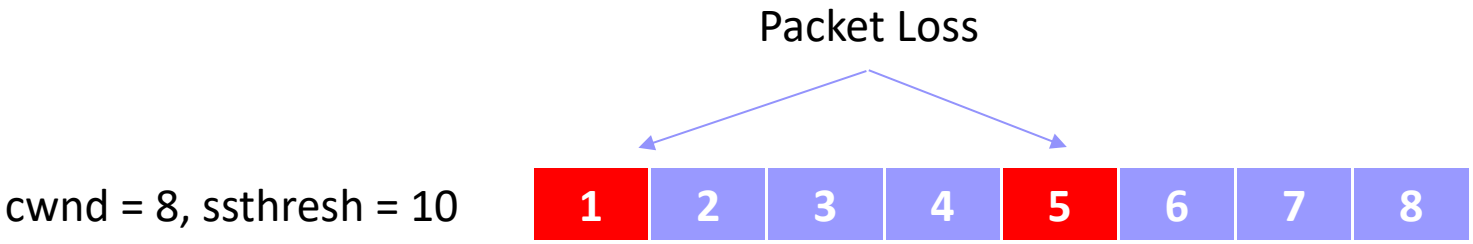
- 快恢复（Fast Recovery）
  - 进入：在快重传之后立即进入
  - 退出：
    - 当对方确认了进入FR前发送的所有数据时，进入Open状态
    - 当触发RTO后，进入Loss状态
  - 在FR内，收到一个ACK：
    - 如果该ACK没有确认新数据，则说明inflight减一，cwnd允许发送一个新数据包
    - 如果该ACK确认了新数据
      - 如果是Partial ACK*，则重传对应的数据包
      - 如果是Full ACK*，则退出FR阶段

*进入FR前的snd_nxt叫做recovery_point (RP)，ACK < RP时为partial ACK，否则为full ACK

数据包重传/丢包恢复

- 什么时候认为发生丢包
  - 超过1个RTT没有收到ACK
    - 快重传：3个dupacks
    - 快恢复：Partial ACK
  - 超时重传定时器触发
    - 认为所有未确认的数据包都已丢失
- 恢复丢包所需时间
  - 快重传：1个RTT
  - 快恢复：n个RTT (n为丢包个数)
  - 超时重传：RTO

快重传&快恢复示意图



	Event	Action
1st RTT	Receive 3 pkts (ACK = 1)	cwnd <- ssthresh <- 4, retrans pkt 1
	Receive 1 pkt (ACK = 1)	cwnd = inflight = 4, do nothing
	Receive 2 pkts (ACK = 1)	It's dupack, send pkt 9, 10
2nd RTT	Receive 1 pkt (ACK = 5)	It's Partial ACK, retrans pkt 5; inflight < cwnd, send pkt 11
	Receive 2 pkts (ACK = 5)	It's dupack, send pkt 12, 13
3rd RTT	Receive 4 pkts (ACK= 11, 12, 13, 14)	It's Full ACK, exit FR, send pkt 14, 15, 16, 17

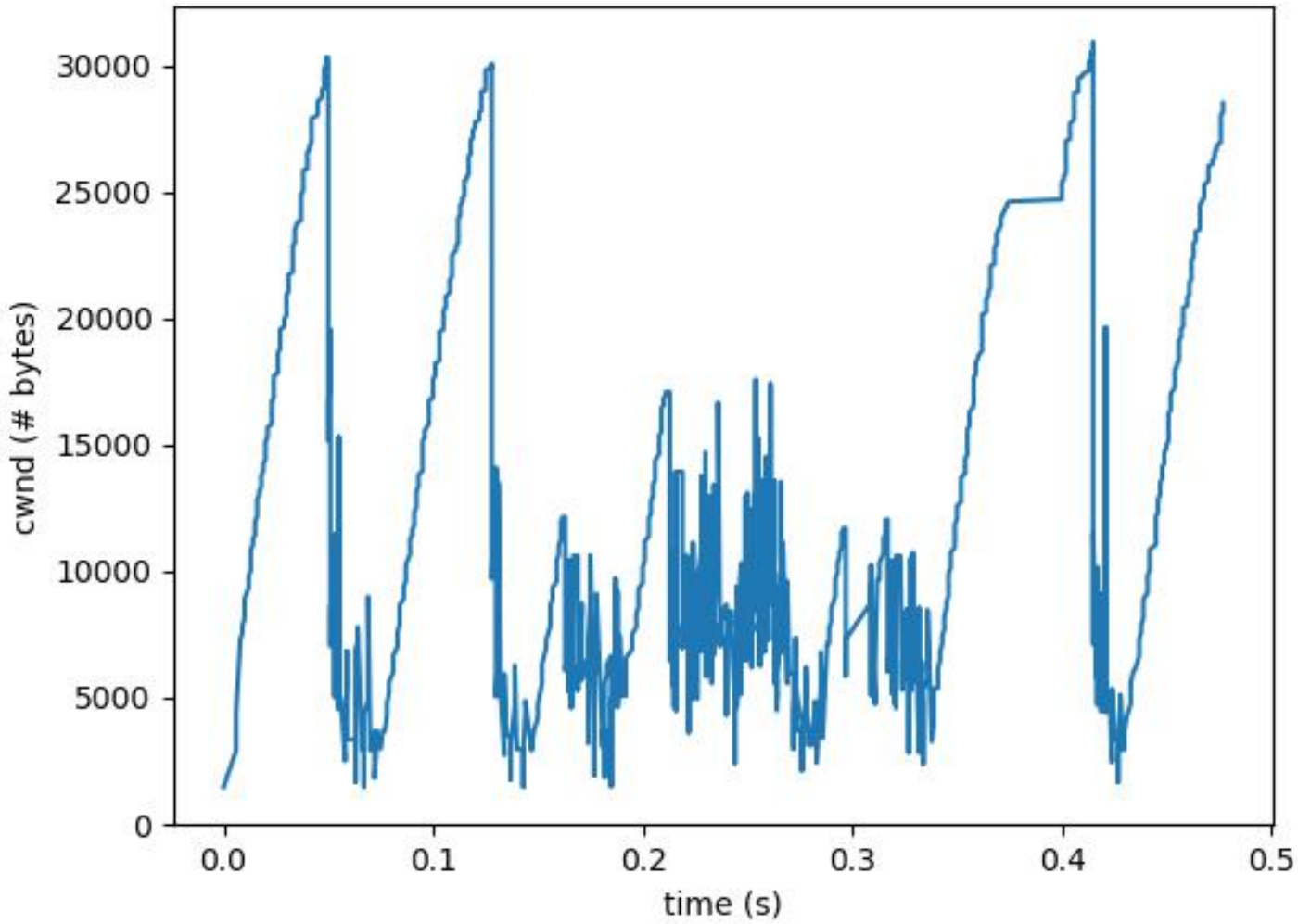
拥塞控制机制实现注意事项

- TCP RFC中cwnd的单位为字节数，Linux协议栈实现中的单位为数据包个数，我们遵从Linux协议栈实现
  - 注意：接收窗口单位为字节，在发送数据包时需要转成包个数
- 窗口大小减半
  - 如果cwnd立即减半，cwnd < inflight，一段时间内不能发送任何包
  - 可以每收到两个ACK，cwnd减1MSS，在一个RTT内窗口能减半，需要添加新的变量（计数器）
- 拥塞避免阶段的窗口增加
  - 在拥塞避免阶段，每个RTT窗口增加1MSS，也可以利用类似上面的计数器实现

TCP实验内容二：拥塞控制

- 执行create_randfile.sh，生成待传输数据文件client-input.dat
- 运行给定网络拓扑(tcp_topo_loss.py)
- 在节点h1上执行TCP程序
  - 执行脚本(disable_offloading.sh , disable_tcp_rst.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 (./tcp_stack server 10001)
- 在节点h2上执行TCP程序
  - 执行脚本(disable_offloading.sh, disable_tcp_rst.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 (./tcp_stack client 10.0.0.1 10001)
    - 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比较两个文件是否完全相同
- 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

实验效果图


30000
25000
20000
15000
10000
5000
0
cwnd (# bytes)
0.0
0.1
0.2
0.3
0.4
0.5
time (s)

附件文件列表

- tcp_topo_loss.py　# 丢包率为2%的拓扑