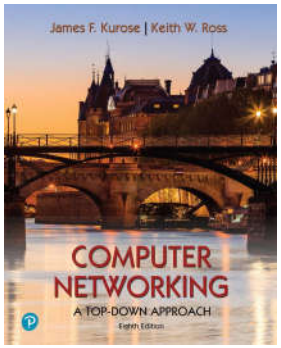
拥塞控制（上）

中国科学技术大学
自动化系　郑烇
部分示意图来自：Jim kurose, Keith Ross教材和ppt



Computer Networking: A Top-Down Approach
8th edition
Jim Kurose, Keith Ross
Pearson, 2020

提纲

1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

提纲

1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

1.TCP拥塞控制概述

- 拥塞非正式定义：太多太快的分组（包含着TCP段以及UDP数据报等）需要网络传输，超过网络（具体来可能是部分节点、链路）的处理能力
- 原因：网络中的某个（某些）路由器队列溢出，开始丢弃分组
- 拥塞的表现，从源端的角度来看
  - 分组（包含着TCP段等）丢失，源端超时了
  - 在源端，收到某TCP段的多个冗余ACK
  - 延迟增加
  - 拥塞情况加速变坏
- 不加控制网络无法使用

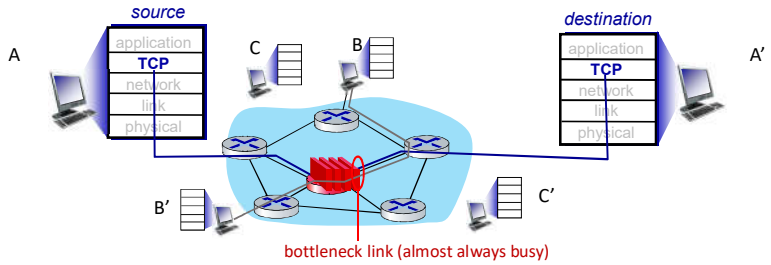
1.TCP拥塞控制概述

- 拥塞控制目标：在网络不拥塞的情况下，尽可能地快速发送（吞吐量）
  - 矛盾目标：不拥塞+吞吐大
  - 不拥塞不是主要目的，是前提；主要目的是大吞吐
  - 系统角度目标：公平
- 关系和位置：Host与网络的关系，非端到端关系（如流量控制）
- 几个关键问题：
  - 拥塞检测：段超时，冗余ACK，ECN，延迟大.....
  - 拥塞时或非拥塞：如何控速
  - 丢失的段什么时机重传
- 控速方法：控制cwnd(在未确认情况下向接收方发送字节数)，速率= cwnd /RTT
  - 不拥塞：增加发送速率
  - 拥塞时：减少发送速率

1.TCP拥塞控制概述

- 为达到目标，具体控制思路：升速和减速最终维持到一个合理的速度
- 在一个相对短的时间，两个主机之间的可达吞吐量基本是个常量
  - 网络的状况发生变化，如某些其他主机对传完了退出竞争，让出带宽，我和对方主机对之间可以传得快些；主机对之间的可用带宽变化成另外常量；
  - 或有一些主机对加入到传输，和我抢带宽，我和对方主机之间的传输只能传得慢些
- IP网没有提供拥塞反馈，各主机不知道什么是合理的速度，动态试凑合理的cwnd
  - 总体控速：发生拥塞时减少，其他时间跃跃欲试增加发送速率；
  - 需要不断按照负反馈控制其cwnd窗口适应网络条件的变化；
  - 各主机对按照该速率进行分布式控制，控制发送速率，间接地协调使用相关联资源，达到各主机对合理使用网络资源的目的

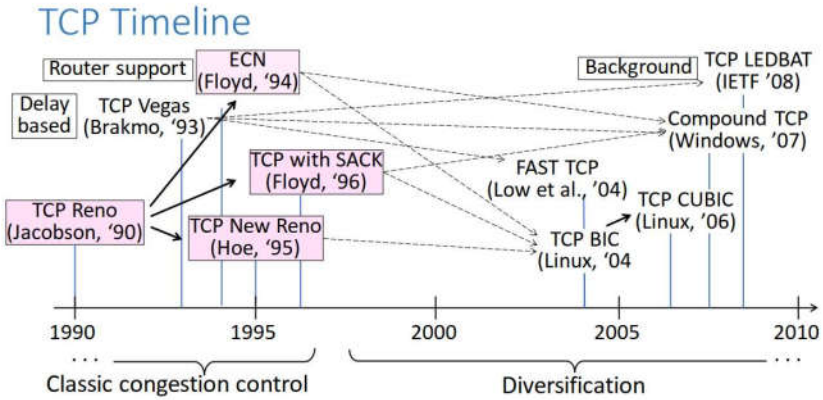
1.TCP拥塞控制概述



提纲

1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

2.拥塞控制算法演化

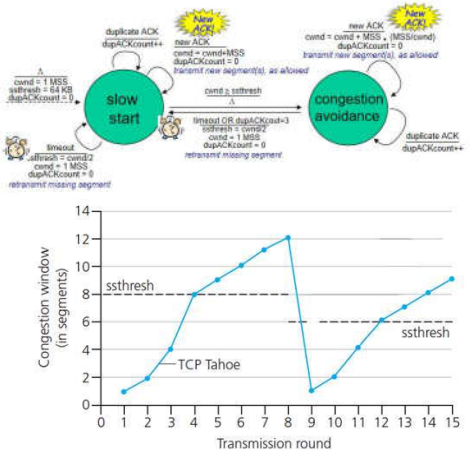


提纲

1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

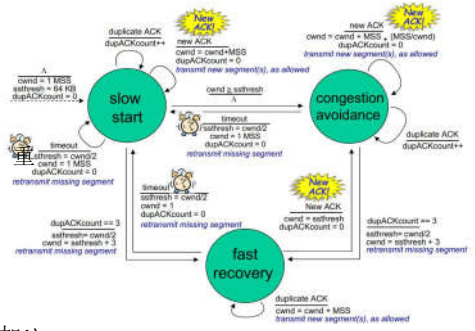
2.经典拥塞控制：Tahoe

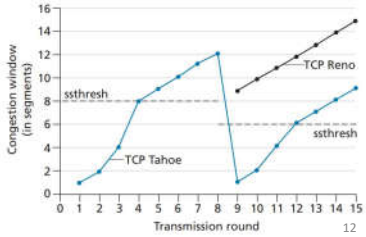
- 拥塞事件：超时+3冗余ACK
- SS状态：
  - 每RTT cwnd加倍（增）
  - 超时或者3冗余ACK：重发，cwnd=1MSS，cwnd的一半为警戒阈值；进入SS状态，
  - 如达到警戒阈值，进入CA状态
- CA状态：
  - 每RTT cwnd+1MSS（增）
  - 超时或者3冗余ACK：重发，cwnd=1MSS，进入SS状态（cwnd的一半为警戒阈值）



3.经典拥塞控制：Reno

- SS状态：
  - new ACK: cwnd=cwnd+MSS(每RTT,cwnd加倍)
  - 超时：ssthresh=cwnd/2,cwnd=1MSS，还在SS状态，发段；
  - 3冗余ACK：快速重传,ssthresh=cwnd/2, cwnd=ssthresth+3+ 进入FR
  - 达到警戒阈值：进入CA状态
- CA状态：
  - new ACK: cwnd=cwnd+MSS/(MSS/cwnd)(每RTTcwnd加1)
  - 超时：ssthres=cwnd/2,cwnd=1MSS，重传
  - 3冗余ACK： ssthres=cwnd/2,cwnd=ssthres+3，重传，进入FR
- 快速恢复状态：
  - 冗余ACK: cwnd=cwnd+1MSS（每RTT倍增，不退出FR）
  - 新ACK：进入CA状态，cwnd=ssthresth（退出FR）
  - 超时：ssthresh=cwnd/2,cwnd=1,入SS,重传



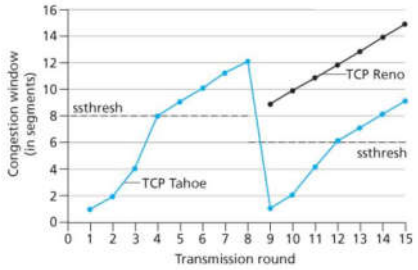


2.经典拥塞控制：Tahoe和Reno

- 经典Tahoe和Reno算法的控制思路：
  - 两个状态SS和CA下的没有拥塞时的增速
    - 警戒阈值上下（上次拥塞时的cwnd一半）分成2个状态
    - 以下：SS状态，离拥塞还早，快速倍增cwnd，增加吞吐量主要矛盾
    - 以上：CA状态，拥塞发生可能性大，线性增cwnd，已经比较快了，避免拥塞是主要矛盾
  - 两个状态SS和CA下的出现拥塞时的降速
    - 超时：ssthresth=cwnd/2；cwnd=1；（重发超时段），进入SS状态
    - 3个冗余ACK：（重发超时段）
      - Tahoe：快速重传，同超时处理
      - Reno：ssthresth=cwnd/2，cwnd=ssthresth+3，通过快速恢复阶段->CA
- Reno的快速恢复：待涉及到的路由器恢复之后，再恢复正常的CA状态
  - Reno收到冗余ACK：cwnd=cwnd+1，有新段发新段，不从FR中退出
  - Reno收到新的ACK：冗余ACK数=0，cwnd=th+3，进入CA
  - Reno超时：重发，ssthresth=cwnd/2，cwnd=1，进入SS

3.经典拥塞控制：Tahoe和Reno

- Reno和Tahoe的差别，收到3个冗余ACK
  - Tahoe：与收到超时事件一样处理：ssthresth=cwnd/2， cwnd=1，快速重传
    - 进入ss
  - Reno：th=cwnd/2，快速重传+（cwnd=th+3）+进入快速恢复状态，CA



提纲

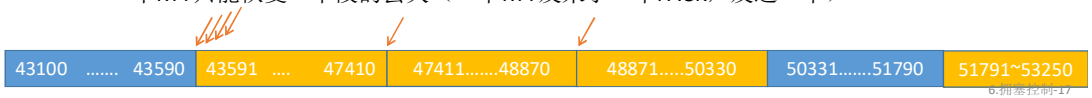
1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno ， SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

3.TCP拥塞控制算法之New Reno

- Reno算法的问题：收到一个新ACK就退出了快速恢复状态
  - 比较适合于单个段丢失情况
- 但经常地、在拥塞时分组是成串被丢弃的，Reno算法问题：
  - 重传冗余ACK对应的段，但是不重传之后的段
  - 后面的那些段丢失会超时,进入到SS阶段
  - 后果：使得发送速率降得过低，cwnd=1
- New Reno算法改进：
  - 发送端记住当时缺少确认的段
  - 收到新确认，重发对应的段
  - 所有待确认的段都重传并得到确认(Recovery ACK)，走出FR状态
  - 当然超时也会从该状态走出进入SS
  - 可以修复多个段的丢失
- New Reno适合场景
  - 多个分组（段）丢失的情况
  - 避免丢失段后续的丢失段超时，进入SS状态来恢复（同时降速）
  - 有效吞吐得以提升

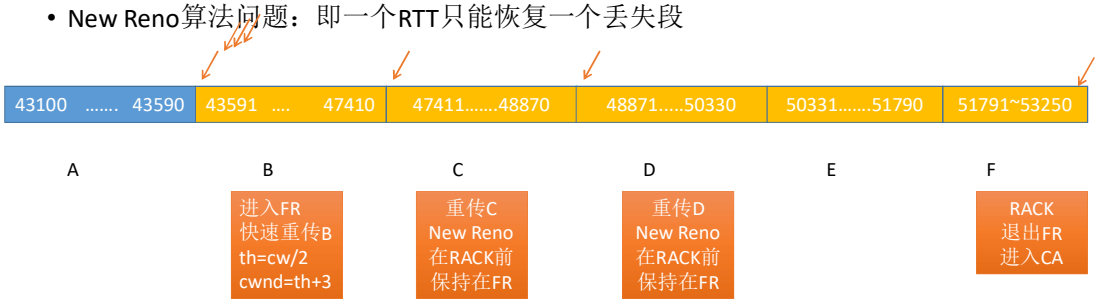
3.TCP拥塞控制算法之New Reno

- 进入快速恢复状态FR：3个冗余ACK，同Reno（ ssthresh = cwnd/2，cwnd = ssthresh + 3）
- 在快速恢复状态FR:
  - 1）再收到冗余ACK：同Reno(cwnd=cwnd+1，条件允许传新段)
  - 2 ）部分确认（PACK）：收到部分新确认，仍保持在快速恢复状态
    - 发送确认后面的段，冗余ACK数量=0， 定时器复位不要超时了， cwnd=cwnd+1（每RTT加倍）
    - 有新段可以发送，发送新的段
  - 3 ）恢复确认(RACK)：收到所有（拥塞时未确认的）段的确认
    - cwnd = ssthresh ， 定时器复位
    - 进入CA阶段
- New Reno（Reno）问题：
  - 一个RTT只能恢复一个段的丢失（一个RTT发来了一个PACK，发送一个）



4.TCP拥塞控制之SACK

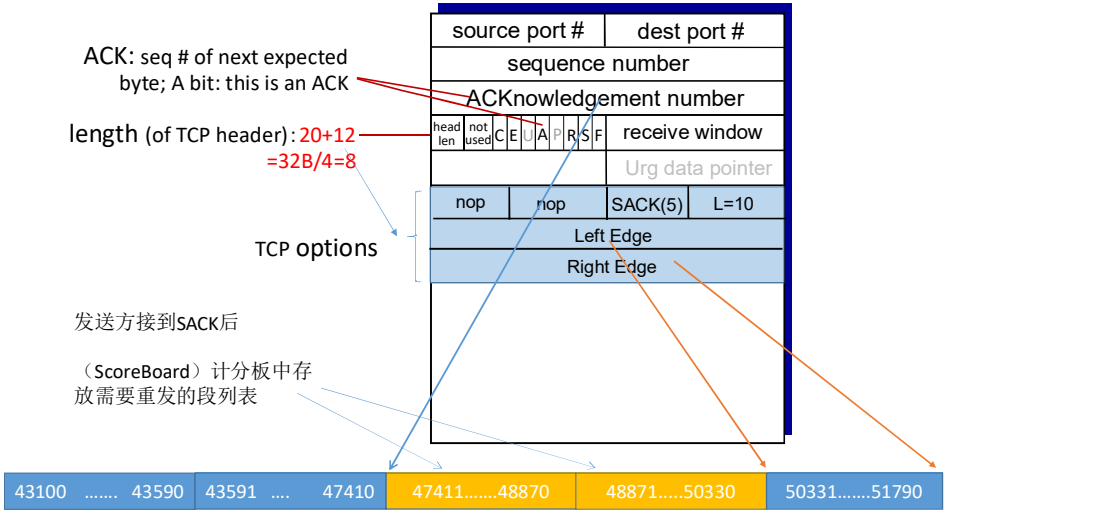
- New Reno算法能够对付段连续丢失的状况
  - New Reno第一个丢失段B确认来了不从FR中走出（不像Reno），RSACK才可以走出FR
  - 从而C段不会超时进入SS(避免Reno算法降速太多的问题)
- New Reno算法问题：即一个RTT只能恢复一个丢失段



4.TCP拥塞控制之SACK

- SACK: 改进
  - 通过SACK，如接收方给出哪些段收到了，哪些段乱序到达了等信息给发送方
  - 在cwnd允许条件下，发送端一次发送多个丢失段，每RTT可以重传多个丢失段，提升效率

问题：接收方如何表述乱序段被接受的情况-SACK



4.TCP拥塞控制之SACK

- 进入FR: 3个冗余ACK，同Reno和New Reno
  - 快速重传ACK后面的段B
  - pipe=待确认的段数量（在管道中已发送出去的段数）th=cwnd/2，cwnd=th+3
  - pipe不能够太满，也不能够太少
- FR状态
  - pipe<cwnd：发送端可先发丢失段(优先计分板中的段），新段，每发一段
    - pipe=pipe+1
  - 每收到确认（冗余否，有无SACK选项），pipe=pipe-x
    - x为从管道中走出的段

4.TCP拥塞控制之SACK

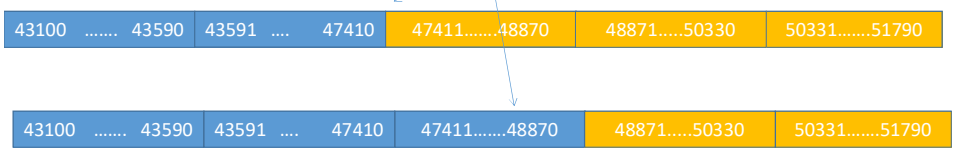
确认	无SACK选项	有SACK选项
冗余ACK	pipe不变，没有段从管道中走出	pipe=pipe-1 SACK指示的新段从管道中走出
PACK（新段得到确认）	pipe=pipe-2，确认的段从管道中走出	pipe=pipe-x
RACK	从FR中退出到CA	

- 冗余ACK(有SACK选项)，pipe=pipe-1

4.TCP拥塞控制之SACK

确认	无SACK选项	有SACK选项
冗余ACK	pipe不变，没有段从管道中走出	pipe-1 SACK指示的新段从管道中走出
PACK（新段得到确认）	pipe=pipe-2，确认的段从管道中走出(第一次的和重传的)	pipe=pipe-x
RACK	从FR中退出到CA	

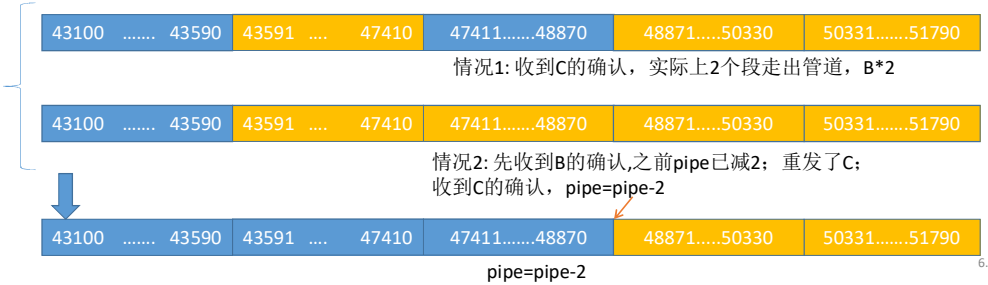
- PACK(无SACK选项)，pipe=pipe-2



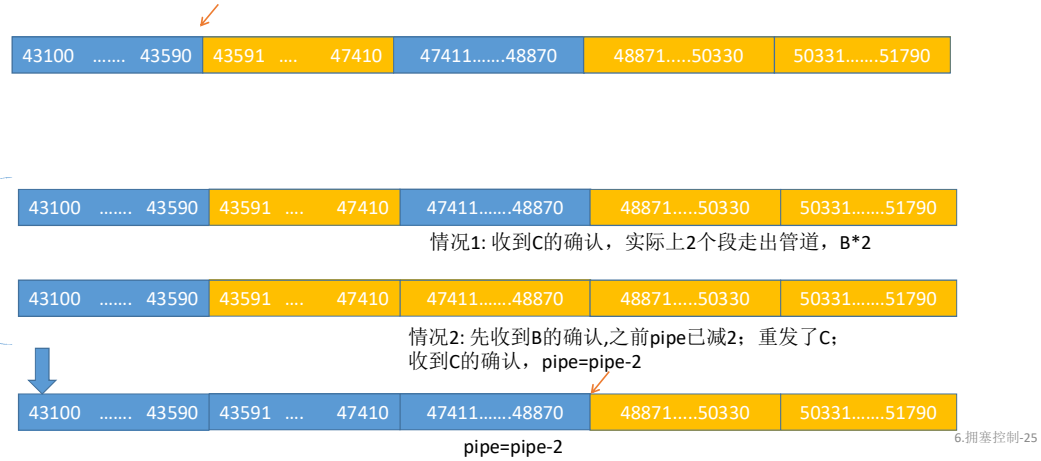
该段确认收到，意味着有2个段从管道走出

4.TCP拥塞控制之SACK

确认	无SACK选项	有SACK选项
冗余ACK	pipe不变，没有段从管道中走出	pipe-1 SACK指示的新段从管道中走出
PACK（新段得到确认）	pipe=pipe-2，确认的段从管道中走出(第一次的和重传的)	pipe=pipe-x
RACK	从FR中退出到CA	

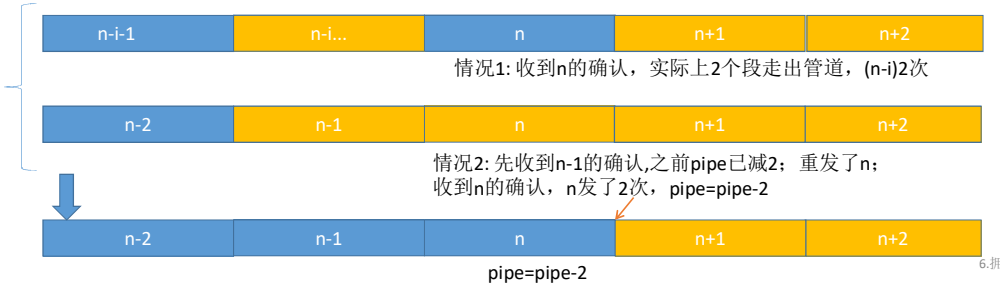


4.TCP拥塞控制之SACK



4.TCP拥塞控制之SACK

确认	无SACK选项	有SACK选项
冗余ACK	pipe不变，没有段从管道中走出	pipe-1 SACK指示的新段从管道中走出
PACK（新段得到确认）	pipe=pipe-2，确认的段从管道中走出(第一次的和重传的)	pipe=pipe-x
RACK	从FR中退出到CA	



4.TCP拥塞控制之SACK

- 接收方SACK：发送方维持ScoreBoard，那些需要重发的段列表
- 发送方维持着已发送但是未确认的段数:pipe
  - pipe<cwnd，先发送计分板中的老段，pipe=pipe+x；
  - 如果pipe<cwnd，在发送新段，pipe=pipe+x；
  - 收到各类确认(冗余否*SACK选项)，pipe=pipe-x
- pipe和ScoreBoard，把什么时候发（pipe<cwnd）和发什么（先ScoreBoard，后新段）解耦
  - New Reno算法：待确认的段数<cwnd，发什么是确定的（累计确认后对应的新段；或者超时重发，三个冗余ACK后面的老段，一头一尾）
- 缺点：需要修改接收端，能发出SACK；发送端修改，根据收到的SACK做相应的动作

提纲

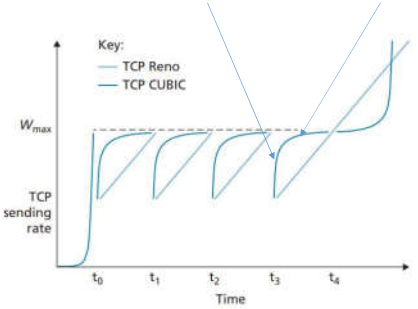
1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

5.改进：CUBIC

- 在某些特定的时间段，两对主机之间的吞吐量基本是个常量，主机对分得的合理带宽相对稳定
  - 对应着：Wmax/RTT
  - 从长期来看不会是一个常量，时常变化
- 经过有限次的迭代，可以比较准确地估计出Wmax和需要经过几轮合适的K让cwnd窗口达到Wmax
  - Wmax/RTT对应着合理的速率，不拥塞时的最大值
  - K是经过几轮能够达到Wmax最合适
- 慢启动和快速恢复阶段和Reno算法一致
- Reno在CA阶段，每RTT cwnd加1MSS的做法过于保守，不利于吞吐量的提升
  - 拥塞控制的目的：不拥塞情况下吞吐量的提升

5.改进：CUBIC

- CBUIC的思想：在Wmax/RTT之下，不拥塞且cwnd尽快接近于Wmax
  - 在CA阶段，CUBIC方式增加cwnd
  - 估计出比较合适是的：K和Wmax
  - T离K越远，增速快；反之，增速慢，慢慢逼近 $cwnd = C \cdot (t-K)^3 + W_{\max}$

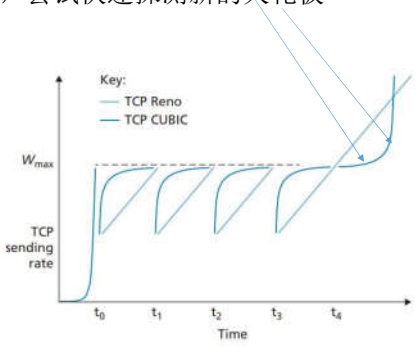


$$K = \sqrt[3]{\beta \cdot W_{\max} / C}$$
$$W_{\max} = \beta \cdot W_{\max}$$

Wmax 上次窗口降低的那个窗口
ß **Wmax**不拥塞的最大窗口，如0.8
C 系数(设定的参数，控制着陡缓)
t 当前时间和上次窗口降低之间的时间差
K 上次丢失事件时更新，计算多长时间达到Wmax

5.改进：CUBIC

- CUBIC的思想：
  - 越过Wmax/RTT一点，比较谨慎地增加，可能是瞬间的优惠
  - 之后增速，尝试快速探测新的天花板



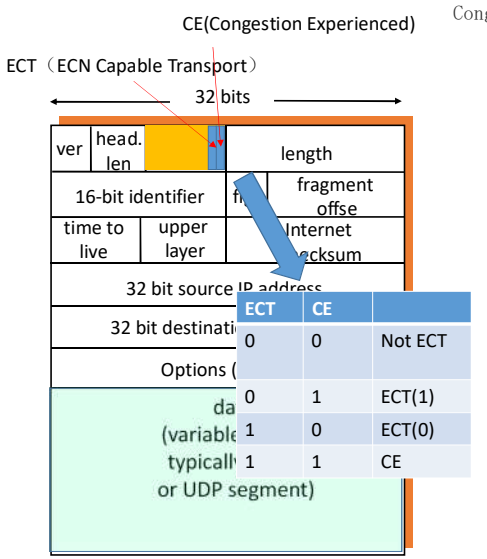
$$cwnd = C \cdot (t-K)^3 + W_{\max}$$

提纲

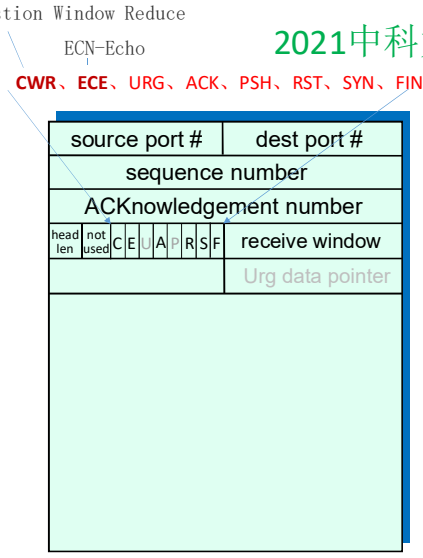
1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

6.网络辅助信息拥塞控制：ECN

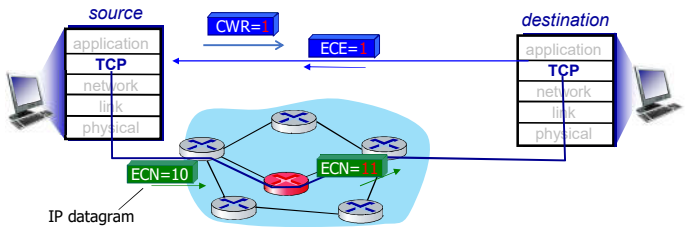
- 拥塞控制的问题：拥塞判断，调速，重传
- 拥塞判断：端系统判断(端系统根据事件来判断拥塞)，网络反馈
- ECN(Explicit Congestion Notification):
  - 网络提供一些有无拥塞的反馈信息
  - 比超时等拥塞事件来的更早，依此进行控制效果更好
- 涉及到了TCP和IP的修改



IP Datagram



TCP Segment



1. 源主机发出去的IP段，相关标志位置位，表示其持ECN，（ECT,CE）=(1,0) or (1,0)
2. TCP段所在IP分组经过拥塞路由器，置位ECN=11
3. 在接收端-发送端的TCP段ACK中ECN Echo=1，接收端后续的ECE都为1
4. 源主机收到ECE=1：降速、指示接收端CWR=1，
   - 本次RTT不再减小窗口
5. 接收端收到CWR=1的段，不再会送ECE=1了
   - 其他的与TCP一起的传输层协议也可以使用ECN,DCCP,DCTCP,DCQCN

提纲

1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

7.基于延迟的拥塞控制：Vegas

- Reno类算法的问题
  - 没有可用带宽的先验知识，不断增加窗口cwnd，只能靠增加窗口拥塞来探测，直到拥塞（ss状态倍增；ca状态持续增加）
  - 反复拥塞降低窗口，带宽利用率不高（拥塞是一种探测可用带宽的方式）
  - 但是这种探测所带来段连续丢失会造成cwnd窗口反复降低
- 延迟比冗余ACK、超时来的更加敏感

- 注：Reno算法RTO是基于500ms计时颗粒度计算的
  - 先3冗余ACK，后超时

7.基于延迟的拥塞控制：Vegas

- 拥塞控制3个问题：检测、控速、重传
- 拥塞检测：丢失超时、3个冗余ACK、网络反馈、延迟变化
  - 超时和3个冗余ACK，网络已经出现拥塞，反应慢

- Vegas：
  - 精确测量RTT，以ms为单位，颗粒度较细
  - 基于延迟变化，能够更早地感知拥塞，更加温和地调节速度，让速率靠近可用带宽

7.基于延迟的拥塞控制：Vegas

SS（慢启动）状态

- 连接建立时，SS，记下第一次BaseRTT
- 2个RTT 倍增cwnd，不会像Reno那样增加的那么快
  - Reno算法不知道可用带宽到底是多少，只有大尺度地倍增
  - 才能够探测到可用带宽
- 2RTT中：
  - 一个RTT，增长期：增加窗口
  - 另外一个测量期：计算期望和实际速度的差
- SS（慢启动）->CA（拥塞避免）
  - 根据测量的RTT，计算期望和实际的速率差大于γ（1），设定阈值
  - 窗口减少1/8；进入CA

7.基于延迟的拥塞控制：Vegas

CA（拥塞避免）状态

- 测量RTT
- 计算期望和实际速率
  - Expected=cwnd/BaseRTT
  - Actual=cwnd/RTT
  - Diff=(Expected-Actual)*BaseRTT （≥0）
- 三种情况，线性增加和减少cwnd，在拥塞路由器缓冲区的段不多不少，让网络保持忙碌状态同时又不拥塞（默认： α=1， β=3）
  - cwnd=cwnd+1，Diff<α时
  - cwnd不变， α ≤Diff≤ β时
  - cwnd=cwnd-1 ， Diff> β时

7.基于延迟的拥塞控制：Vegas

重传

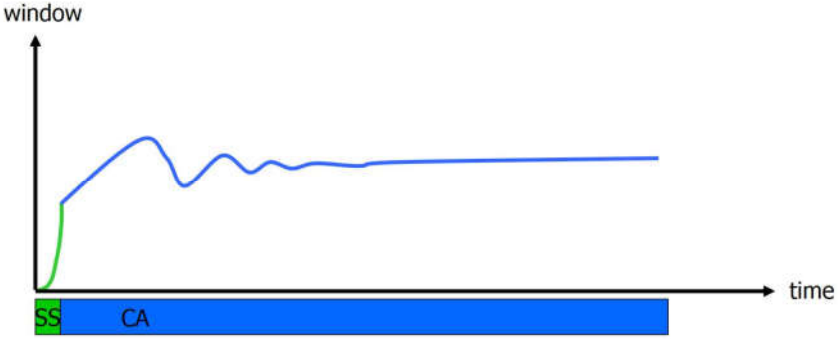
- 时机1：冗余ACK，且段RTT超过RTO（还没有收到3个冗余ACK）
- 时机2：重发之后的第一个第二个，非冗余ACK，如果RTT超过RTO（没有收到3冗余ACK）

- 重发之后窗口减少1/8

- 在一个RTT之内不会减少两次，像Reno算法那样
  - 减少之后的 丢失或者拥塞不会再次减少cwnd

7.基于延迟的拥塞控制：Vegas

- Vegas算法由于以下问题，在互联网上使用的不多
  - 路径变化带来的RTT变化，误以为是拥塞，做出误动作
  - 与其他CUBIC、Reno算法协同工作的不公平性（敏感且温和）



提纲

1. TCP拥塞控制概述
2. 拥塞控制算法演化
3. 经典拥塞控制：Tahoe和Reno
4. 改进：New Reno 、SACK
5. 改进：CUBIC
6. 网络辅助信息拥塞控制：ECN
7. 基于延迟的拥塞控制：Vages
8. 总结

8.总结

- 拥塞：各涉及到的主机往网络中注入的分组速率过快，超过某个（些）路由器的处理能力
- 表现：超时、冗余ACK、延迟大、ECN
- 拥塞控制：在网络不拥塞情况下，尽可能发送发送得快
- 主要问题：
  - 探测拥塞
  - 最核心的问题：控制速度，估计当前比较合理的cwnd
  - 重传机制