TCP拥塞控制

教材目录

3.3 数据链路协议：可靠数据传输

- 停等协议
- 滑动窗口协议：GBN和选择重传

6.2.1 TCP概述

6.2.2 TCP段格式

6.2.4 TCP可靠传输

6.2.5 TCP流量控制

6.2.3 TCP连接管理

TCP可靠数据传输

6.2.7 TCP计时器

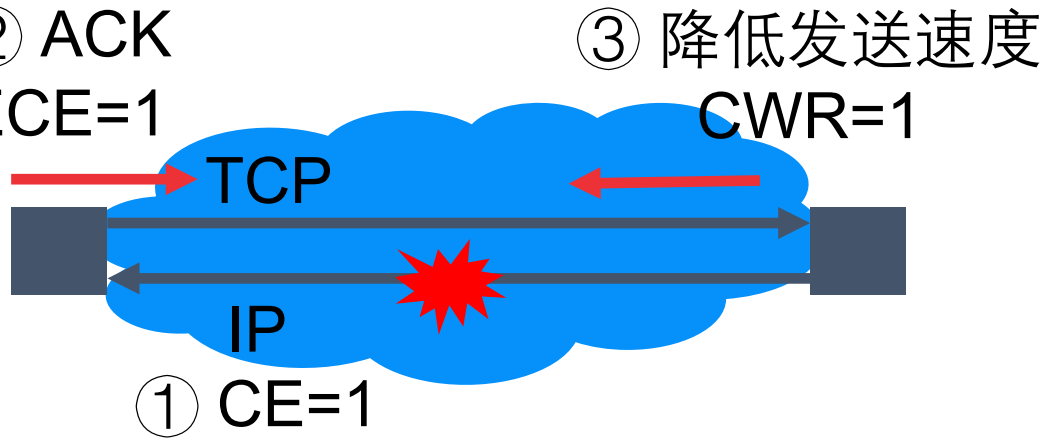
6.2.6 拥塞控制

6.3.1 UDP

拥塞控制(Congestion Control)

数据偏移	保留(4b)	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN

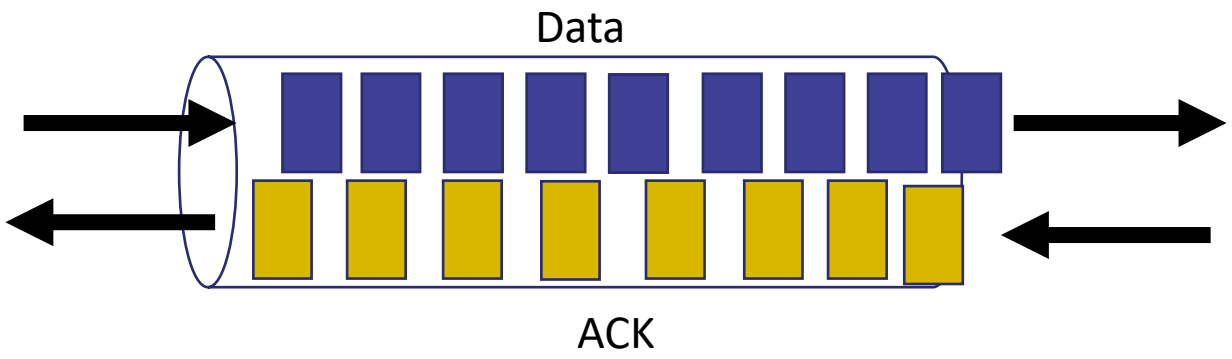
- 拥塞：在某个时刻多个发送者发送的数据太多以致网络的某个部分的容量无法容纳这么多的分组
- 拥塞导致：
  - 分组丢失：路由器的缓冲区满而丢弃分组
  - 长延迟：分组在路由器处排队
- 拥塞控制：
  - 端到端的拥塞控制： 端系统判断是否出现拥塞并采取相应的动作
  - 基于网络(network-assited)的拥塞控制：网络提供拥塞反馈机制，端系统根据拥塞指示采取相应的动作
    - 拓展： RFC 3168 ： Explicit Congestion Notification (ECN)
      - 路由器将经过的IP分组的头部的Congestion Experience (CE)比特设置为1
      - 接收者在收到数据后如果其中CE=1时，发送的TCP ACK中通过**ECN-Echo**比特设置为1 (ECE=1)通知发送者出现了拥塞
      - TCP ACK可能丢失，后续发送的ACK中ECE=1，直到接收者收到一个新的**CWR (Congestion Window Reduced)=1**的TCP段为止



拥塞控制与流量控制

- 流量控制：防止发送过快以至于另一方缓冲区用完的情况
  - 接收窗口rwnd：接收方在ACK中给出其空闲的缓冲区大小
  - 接收窗口限制了发送方的发送速度
- 拥塞控制：防止以超过Internet处理能力的速度发送， 1988年Van Jacobson引入了TCP拥塞控制
  - 拥塞窗口cwnd：估计的网络容量的大小
  - 拥塞窗口限制了发送方的发送速度
  - TCP的数据传输是ACK驱动（ACK-Clocked）的：接收者每次（在采用Delay Ack时每两次）收到一个TCP数据段时发送一个ACK
  - 发送者根据收到的ACK的情况来更新cwnd：
    - 网络出现拥塞时减少cwnd
    - 网络比较空闲时增加cwnd
- TCP的发送受到流量控制和拥塞控制的制约：

  LastByteSent – LastByteAcked <= min(cwnd, rwnd)



拥塞事件(Loss Event)

TCP**假设分组丢失的主要原因**是由于**网络拥塞（缓冲区满）而丢弃**分组，而不是由于链路上的差错导致的分组丢失

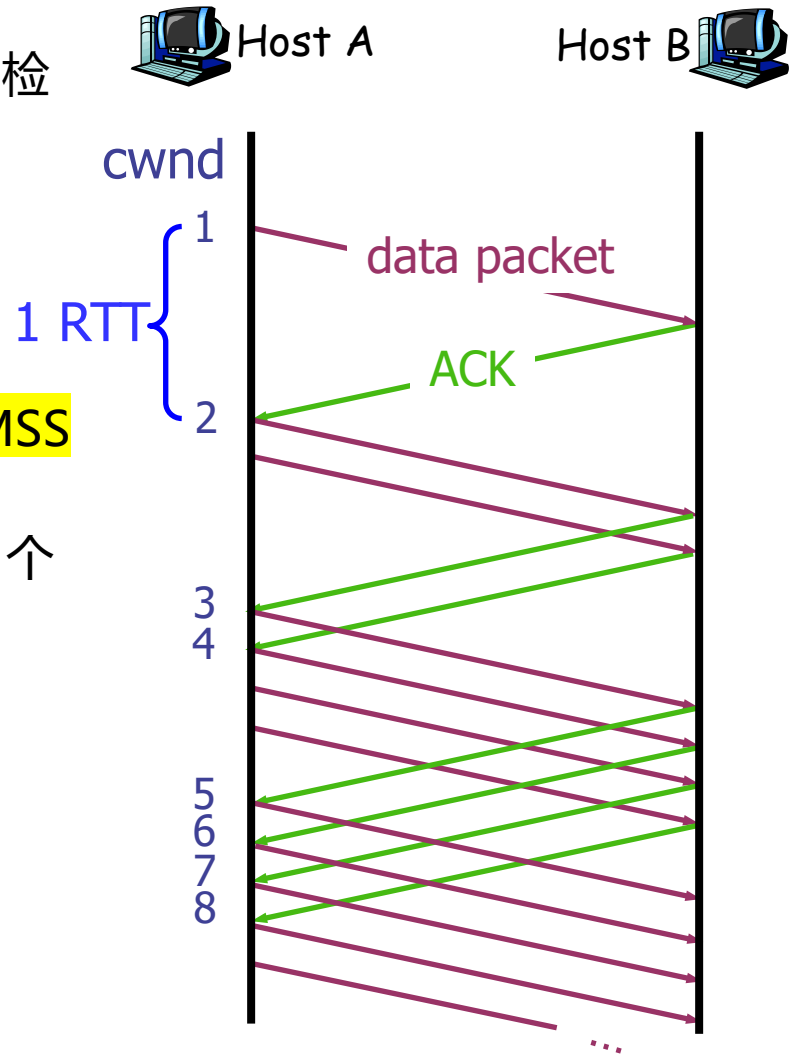
- **拥塞事件（Loss Event）**
  - 超时： **(严重)拥塞**出现，发送者需要大幅降低发送速度
  - 3个重复ACK：说明有分组丢失但也有3个分组到达接收者，**轻微拥塞或者已从拥塞恢复**
- 网络的**容量动态变化**：随着新连接的建立和老连接的释放，容量也不断变化
- 拥塞控制可以分为两个部分：
  - (在连接刚刚建立或者出现严重拥塞时）**探测网络的可用容量cwnd**（慢启动）
  - 网络容量变化时**动态调整cwnd（AIMD)**

$$rate = \frac{cwnd}{RTT} \text{ Bytes/sec}$$

慢启动 Slow Start

- 新建的TCP连接，刚开始以**较慢的速度发送**，然后迅速增加发送速度，直到检测到拥塞出现为止
- 初始拥塞窗口cwnd=**初始窗口IW**
  - 最初的规范建议初始窗口IW为1个MSS
  - **RFC 5681 TCP Congestion Control** 建议：IW为2到4个MSS
  - 拓展 RFC 6928建议：初始窗口提高到最多10个MSS（具体多少根据MSS的大小选择)
  - 连接建立阶段的ACK不改变拥塞窗口。如果SYN有重传，则IW设置为1个MSS
- 连接建立后，每收到一个**新的ACK**，拥塞窗口增加1个MSS
  - 如果收到一个ACK，表示有一个分组到达了接收者
  - 每个RTT之后，拥塞窗口翻倍

```
# 拓展 IW为4k字节以上
if SMSS > 2190:
  IW = 2 * SMSS
elif SMSS > 1095 and SMSS <= 2190:
  IW = 3 * SMSS
else: # SMSS <= 1095
  IW = 4 * SMSS
```

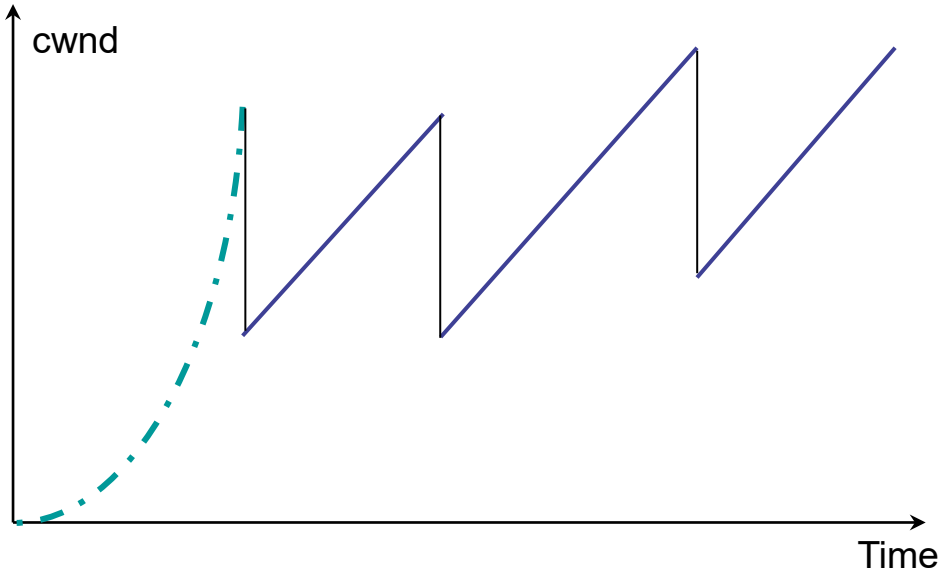


cwnd ← cwnd + 1 MSS (for each ACK)

动态调整拥塞窗口：理想情形

AIMD（Additive Increase/Multiplicative Decrease）

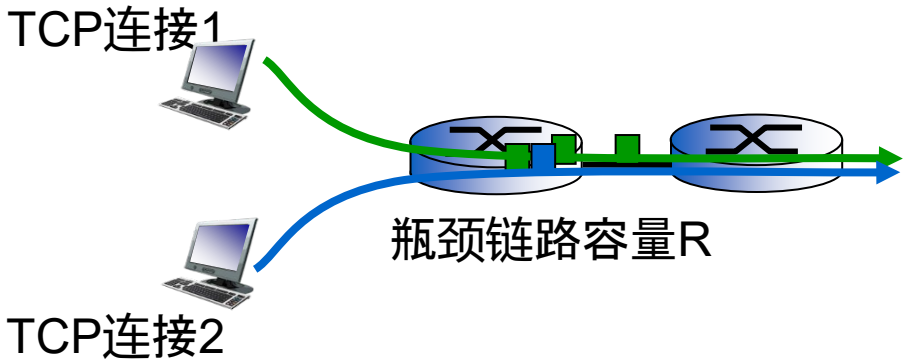
- 当拥塞窗口达到网络的容量之后，路由器将丢弃分组
  - 估计过高，(迅速）减少拥塞窗口，拥塞窗口为原来的1/2，即**倍数减少（MD）**
  - 可能减少得太多，继续(缓慢）增加拥塞窗口，每RTT拥塞窗口加1个MSS，即**线性增加（AI）**
  - 接着拥塞窗口达到网络容量，拥塞窗口减半
  - 如此继续，拥塞窗口在网络容量附近抖动
- 注意：TCP连接的容量是动态变化的
  - 一条连接关闭，可用容量增加
  - 另外一条连接建立，可用容量减少



拓展：AIMD的理论基础

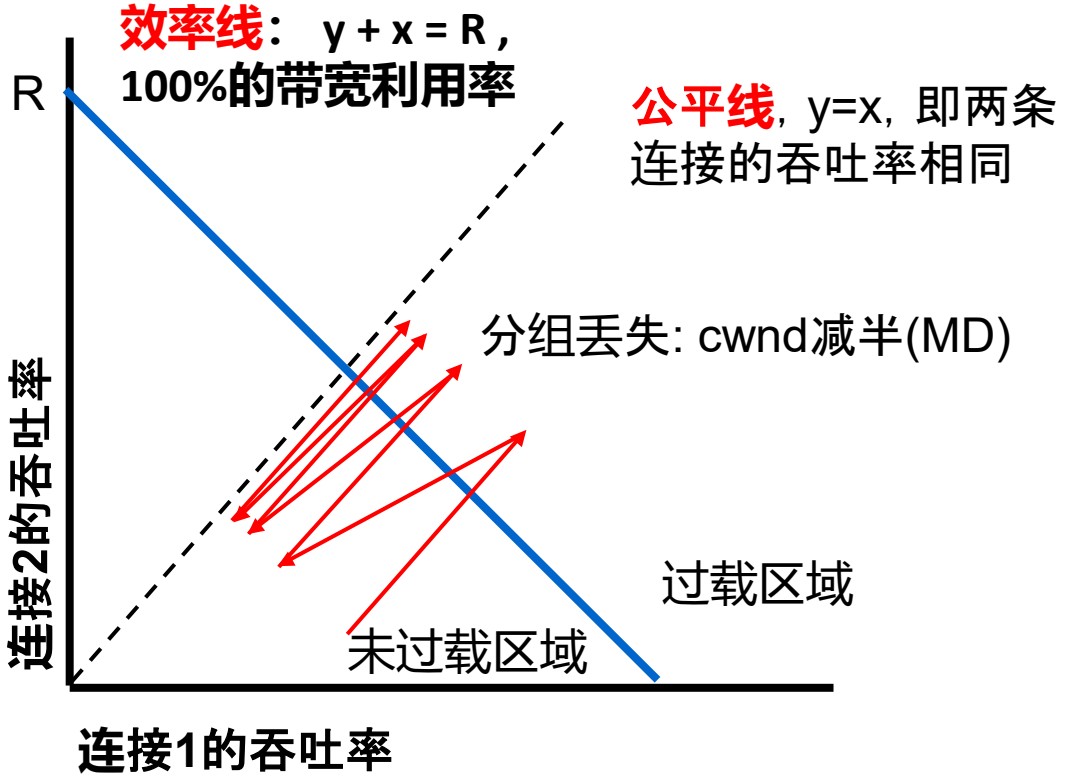
考虑效率和公平性：

- 如果K个TCP连接共享一个容量为R的瓶颈链路，每条连接的平均吞吐率应该为R/K



假设初始两个连接的cwnd分别为(x0,y0)，对于同一个事件进行响应

- 加数增加AI（cwnd += b，b= 1）：
  - 变为（x0+b, y0+b)
  - 斜率为1的增加
- 倍数减少MD（cwnd *= a, a =1/2）：
  - 变为(ax0, ay0)
  - 按比率减少，经过原点



两条TCP连接的相图:
纵横坐标分别为两个连接的吞吐率

动态调整拥塞窗口：实际情形

- 发送者通过**发送方超时**检测到网络出现严重拥塞时：
  - 相比网络拥塞出现的时刻**已经滞后**，网络容量也可能动态变化
  - 此时**管道可能已经空**，因而没有ACK来驱动新的分组发送
  - 需要重传分组，并且**重新探测网络的容量**
- 发送者重新开始慢启动，cw=丢失窗口(Loss Window)，LW为1个MSS
- 发送者可以估计到在RTO超时时刻的容量为拥塞窗口的一半
  - 在接近该容量时拥塞窗口增加的速度减慢
  - 引入**慢启动阈值ssthresh=超时时拥塞窗口的一半**：拥塞控制的哪个阶段?
    - 拥塞窗口小于ssthresh，则为慢启动阶段，拥塞窗口指数增加
    - 拥塞窗口大于(等于) ssthresh，则为拥塞避免阶段，拥塞窗口线性增加
    - 拥塞窗口=ssthresh，标准中可采用慢启动也可拥塞避免，实践中一般采用拥塞避免

动态调整拥塞窗口：拥塞避免

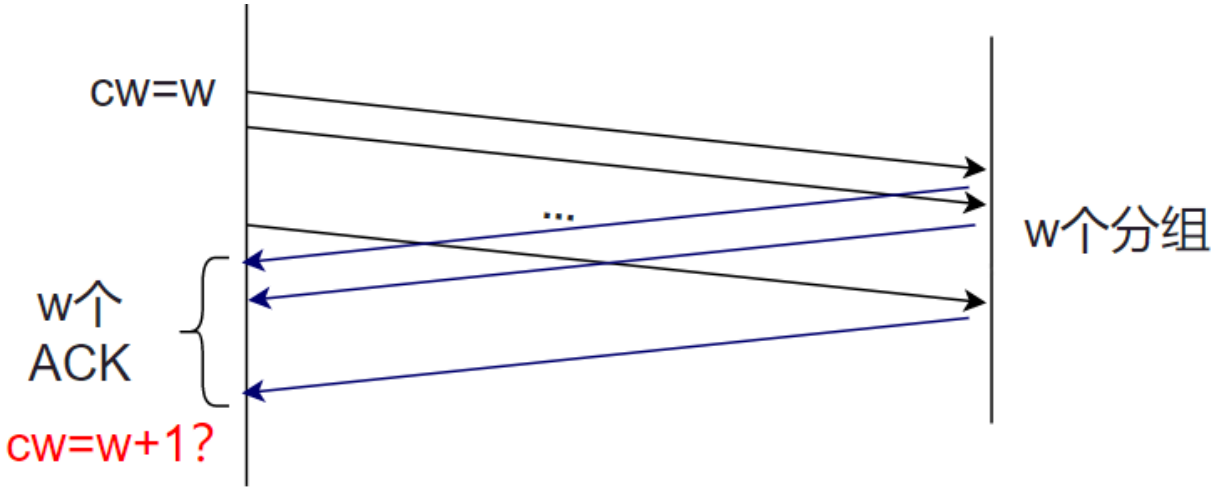
拥塞窗口超过(或等于)慢启动阈值时，进入拥塞避免(Congestion Avoidance)，(缓慢）增加拥塞窗口

- 线性增加：每RTT拥塞窗口最多增加1个MSS

  cwnd ← cwnd + MSS　per RTT

- 实现1：RFC 5681
  - 一个RTT会收到cwnd/MSS个ACK，总共增加1个MSS
  - 每收到一个新的ACK

    cwnd ← cwnd + MSS*MSS/cwnd　per ACK



拓展：Appropriate Byte Counting（ABC）

- ACk Division：收到一个TCP段，发送多个新的ACK，每个ACK确认其中一部分数据

Appropriate Byte Counting（ABC）， Linux系统缺省采用此方法

- 在拥塞控制阶段，目标：**每RTT拥塞窗口最多增加1个MSS--> 确认了cwnd大小的数据后增加1个MSS**
  - bytes_acked: 记录了上次更新拥塞窗口之后累计收到的所有ACK新确认的字节数
  - 如果bytes_acked >= cwnd，则：

```
bytes_acked ← bytes_acked - cwnd
cwnd ← cwnd + MSS
```

- 在慢启动阶段
  - 每收到一个新的ACK时，ack.bytes_acked为这个ACK新确认的字节数

```
cwnd ← cwnd + min(ack.bytes_acked, MSS)
```

动态调整拥塞窗口：超时和重开空闲连接

- 如果RTO计时器超时，意味着**严重拥塞**
  - 慢启动阈值ssthresh设置为当前拥塞窗口的一半

    $ssthresh \leftarrow cwnd/2$
  - cwnd设置为丢失窗口(Loss Window)，LW为1个MSS

    $cwnd \leftarrow 1\ MSS$
- 连接空闲较长时间时：
  - TCP模块没有办法利用ACK来探测网络的容量
  - 仍然按原有的拥塞窗口大小发送数据，可能会造成网络的拥塞
- 如果TCP空闲超过一段时间（**重传超时的时间**），在重新开始传输数据时，进入慢启动过程
  - 慢启动阈值ssthresh设置为当前拥塞窗口的一半

    $ssthresh \leftarrow cwnd/2$
  - 拥塞窗口cwnd设置为重启窗口(restart window)，一般为IW

    $cwnd \leftarrow 1\ MSS$

TCP拥塞控制

cwnd的单位为分组
cwnd的单位为字节

最早引入的TCP拥塞控制：（新建连接或超时时）**慢启动+拥塞避免**

慢启动（Slow Start）

- cwnd指数增加
  每RTT: cwnd ← 2 x cwnd
- 初始ssthresh设置为较大的值(=接收窗口)，
  cwnd ← 1
  cwnd ← 1*MSS --Initial Window
- 每收到新的ACK，
  cwnd ← cwnd + 1
  cwnd ← cwnd+ 1*MSS
- 当cwnd >= ssthresh时进入CA

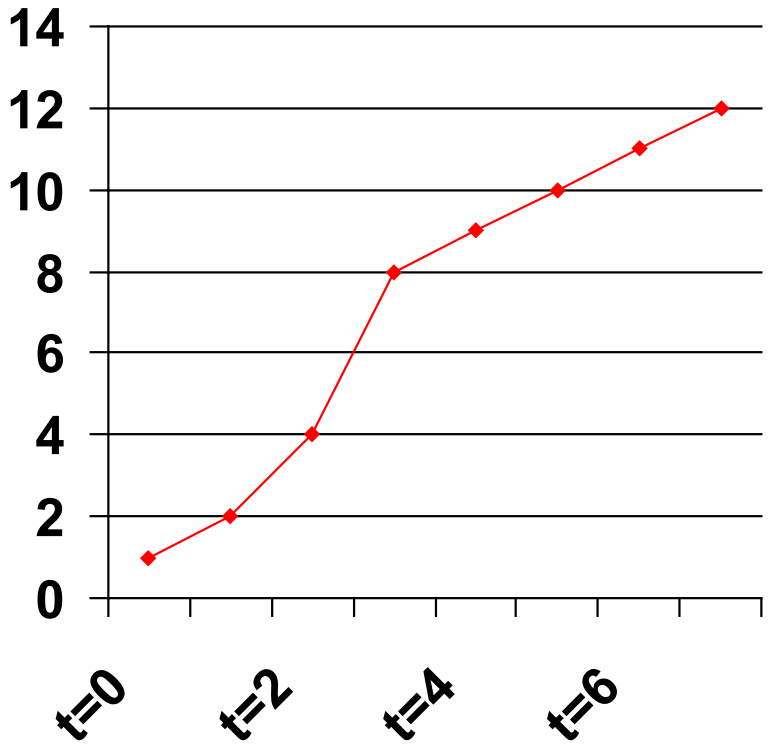
拥塞避免(Congestion Avoidance)： Addititive Increase

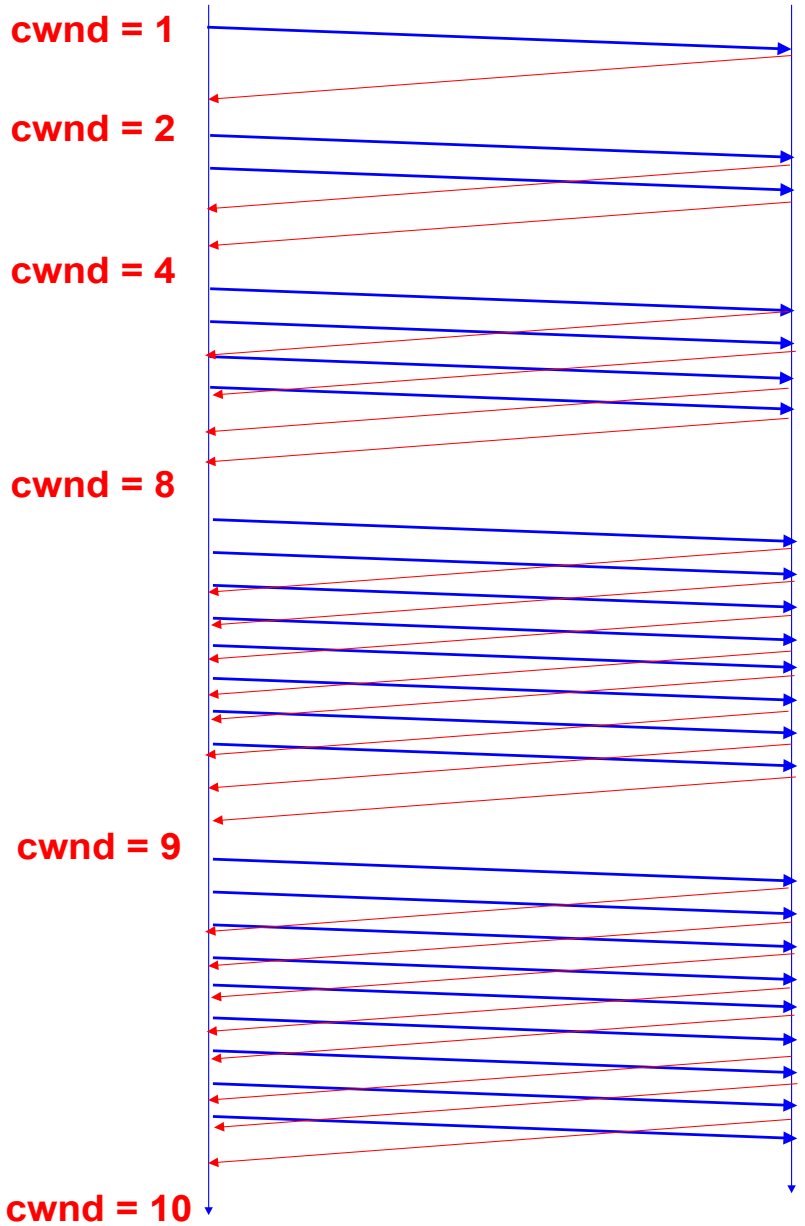
- cwnd线性增加
  每RTT: cwnd ← cwnd + 1
  cwnd ← cwnd + MSS
- 转换为： 每收到新的ACK
  cwnd ← cwnd + 1/cwnd
  cwnd ← cwnd + MSS*MSS/cwnd

超时(Timeout)： 倍数减少(Multiplicative Decrease)
空闲超过重传超时时间时：
ssthresh = cwnd/2
cwnd ← 1
cwnd ← 1*MSS --- Loss/Restart Window
重新进入慢启动

TCP拥塞控制：窗口变化

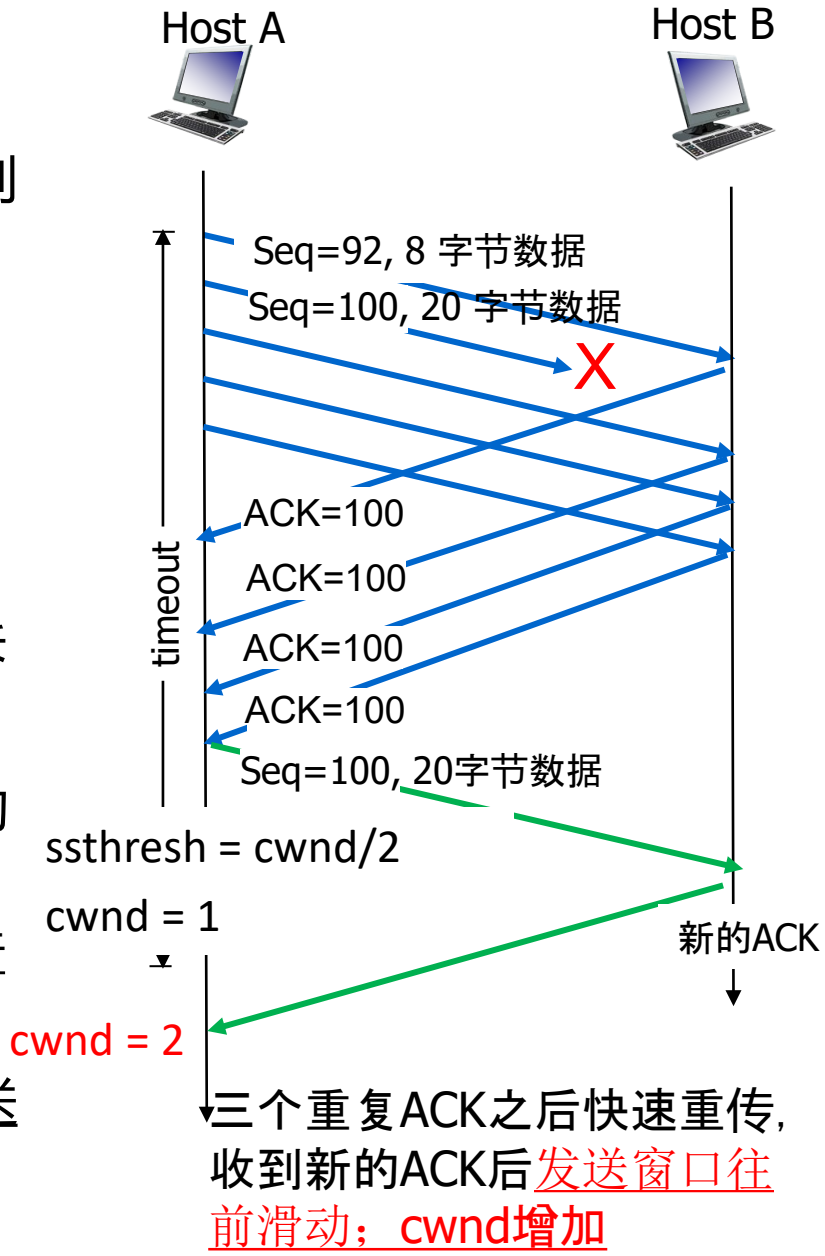
ssthresh = 8





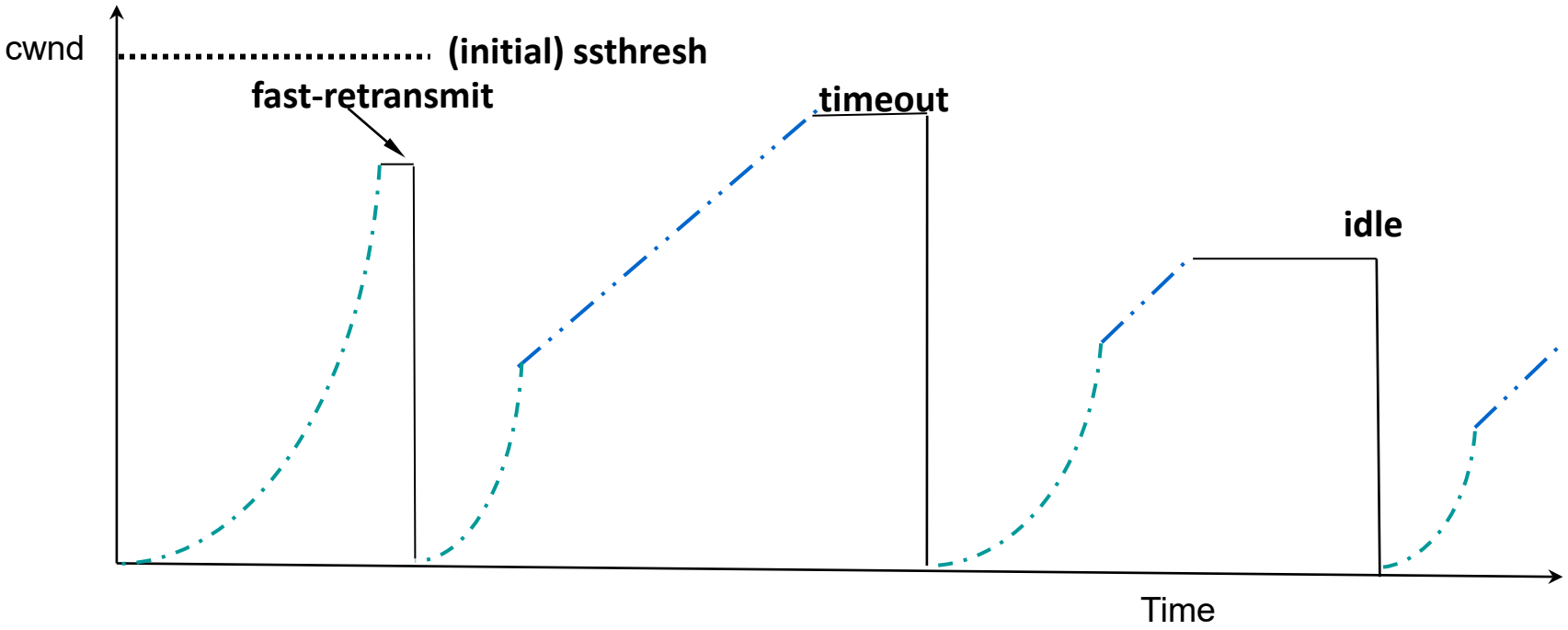
快速重传(Fast Retransmit)

- 重传超时一般比实际的RTT要长一些: 等待超时才能重传丢失TCP段
- 接收者：收到失序TCP段也要发送重复ACK（最后一个收到的已按序到达的TCP段的ACK)
- 发送者经常连续发送多个TCP段，分组丢失时会触发多个重复ACK
  - 收到一个重复的ACK，有两种可能
    - 被确认的TCP段后面的TCP段被暂时延迟，但最终会到达，尽管是失序到达：无需重传
    - 该TCP段丢失：一个重复的ACK可以看作是一个预警，它告诉源端一个TCP段已经丢失了而必须重传
- TCP发送方收到3个重复ACK(即共收到同一报文段的4个ACK)：随后的TCP段已丢失的可能性就很大，**立即重传该丢失的TCP段**
  - 进入**慢启动阶段**，在新的ACK回来或超时前忽略后续的重复ACK（不进行快速恢复）
- 重传的TCP段到达接收者后，会对所有迄今为止按序收到的报文段发送一个累加ACK



TCP Tahoe

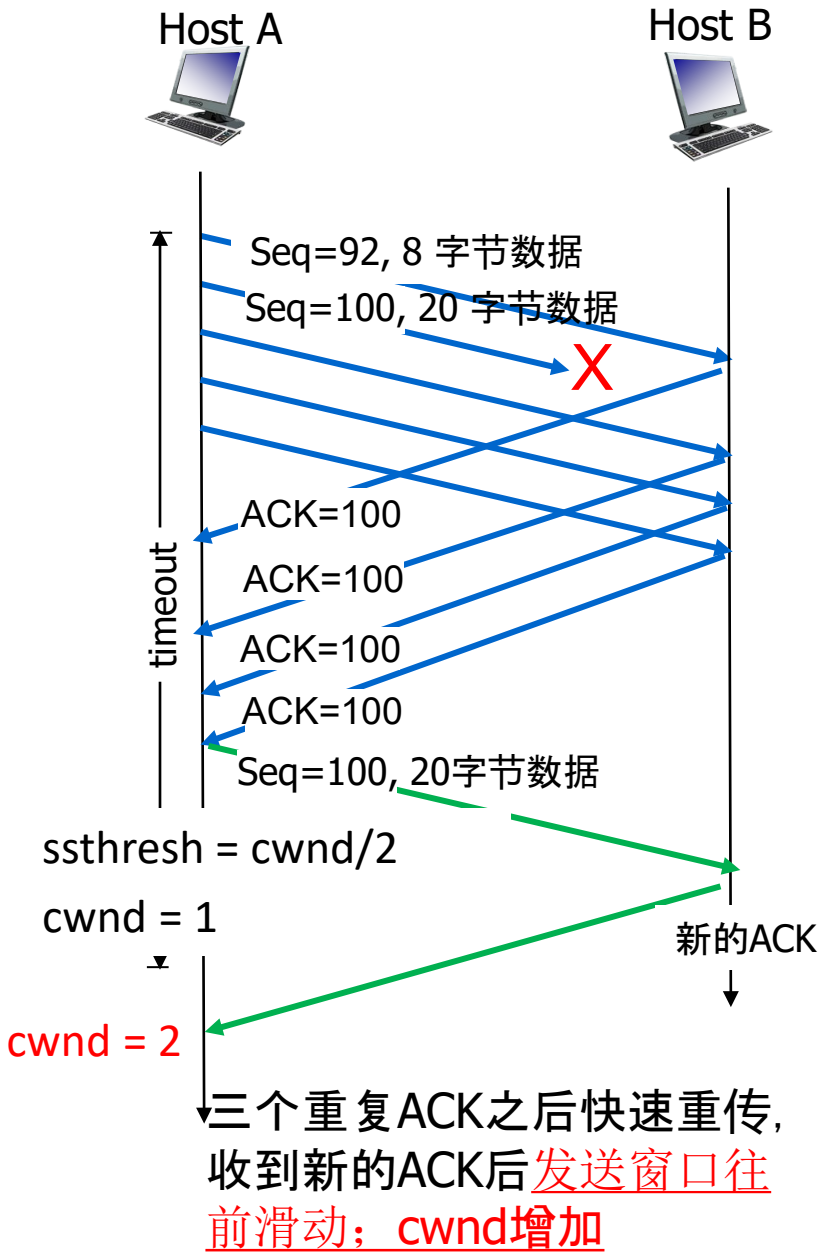
- **慢启动+拥塞避免+快速重传（**快速重传后进入慢启动阶段)



TCP Tahoe的拥塞窗口变化示意图

快速恢复(Fast Recovery)

- 快速重传后采用慢启动时：
  - 在该RTT仅仅重传第一个认为丢失的TCP段
  - 减慢发送速度到1MSS/RTT，再指数增加
- **但是：**
  - 收到重复的ACK不仅表示有TCP段丢失，而且：
    - 接收者只有当收到一个TCP段时才产生重复ACK，该TCP段已经离开网络进入用户缓冲区，即**有一个TCP段不再占用网络资源**
  - 会有更多的（重复的）ACK到来，发送者有机会继续传输新的TCP段
  - 发送者在快速重传后的RTT期间：
    - 应该以网络容量（快速重传前的拥塞窗口）的一半的速度发送



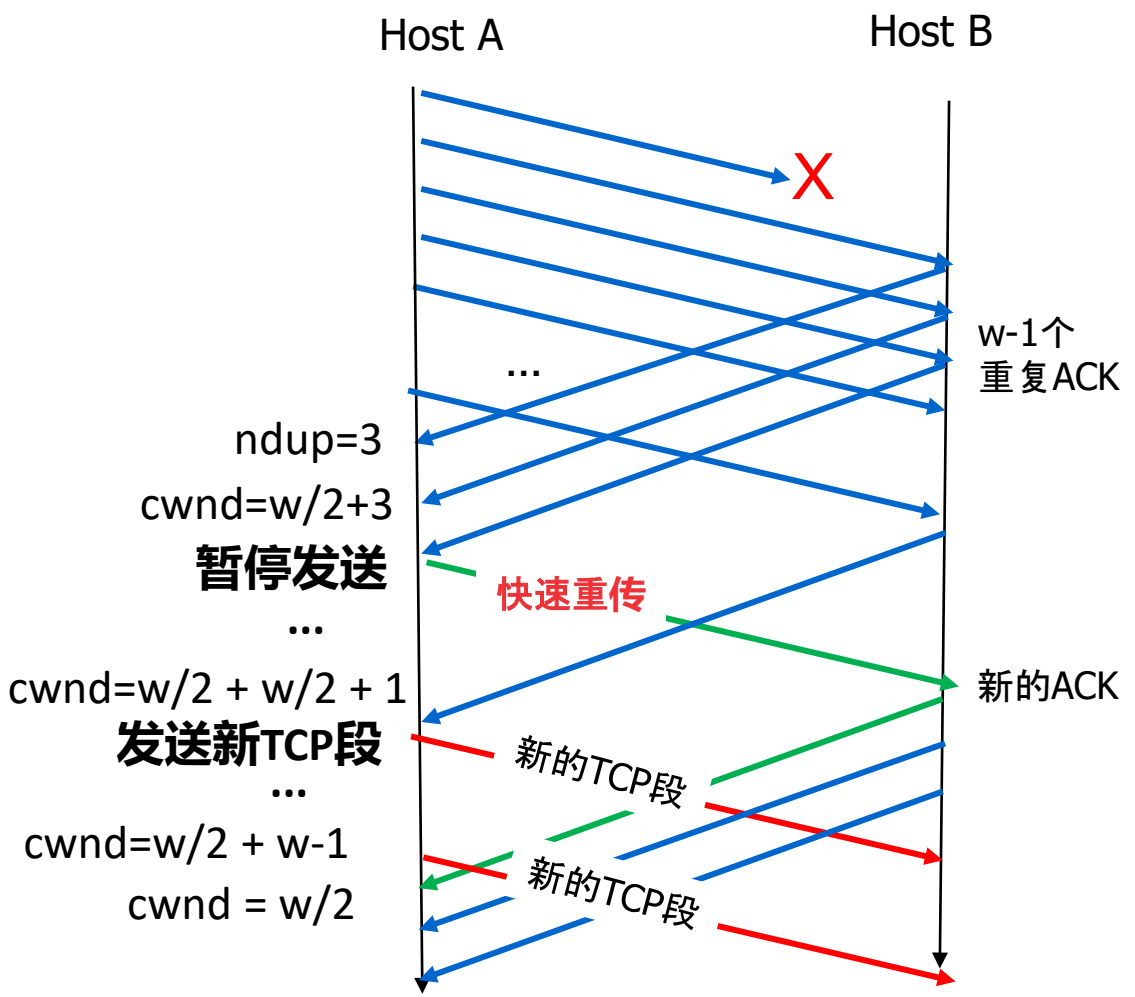
快速恢复(Fast Recovery)

ndup记录最近收到的重复ACK的个数

- 当收到第三个重复ACK（即ndup == 3）时：
  - 快速重传丢失的TCP段
  - 设置**ssthresh = max(cwnd/2, 2*SMSS)**
  - 设置**cwnd = ssthresh + ndup*MSS**（已经离开网络并且被缓存）→inflating
- 每次收到更多的重复ACK(对同一TCP段)时，更新ndup
  - 拥塞窗口加1：**cwnd += MSS**;

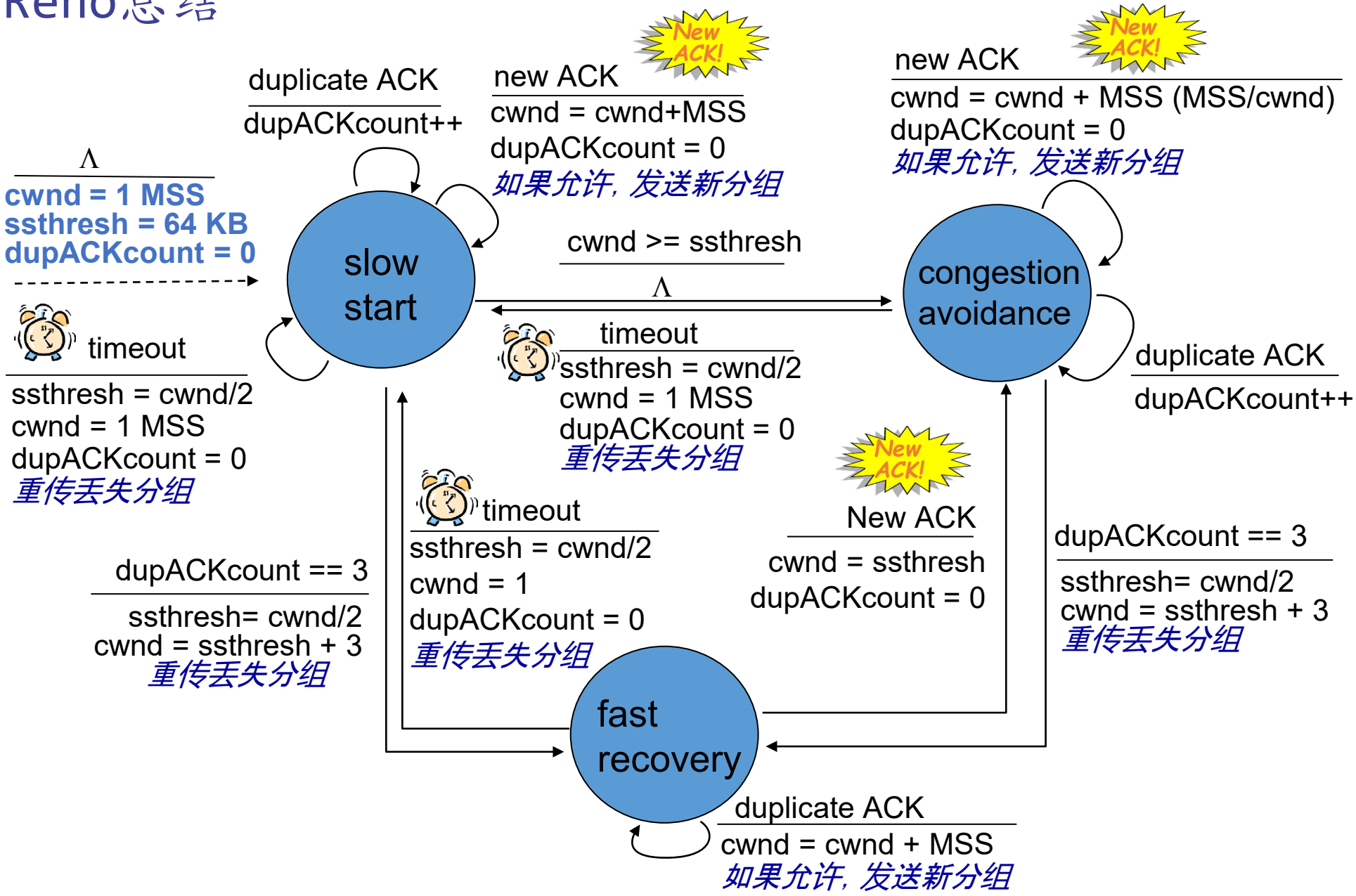
    →inflating **(cwnd = ssthresh + ndup* MSS)**
  - **如果cwnd允许**，传输新的分组 **w/2-1**
- 当确认新数据的下一个ACK(即丢失报文段及其后各报文段的累加确认)到达时:
  - 设置**cwnd = ssthresh**， →deflating (ndup = 0)
  - **进入拥塞避免阶段**
- 如果期间RTO超时，慢启动：

  ssthresh = cwnd/2; cwnd = MSS



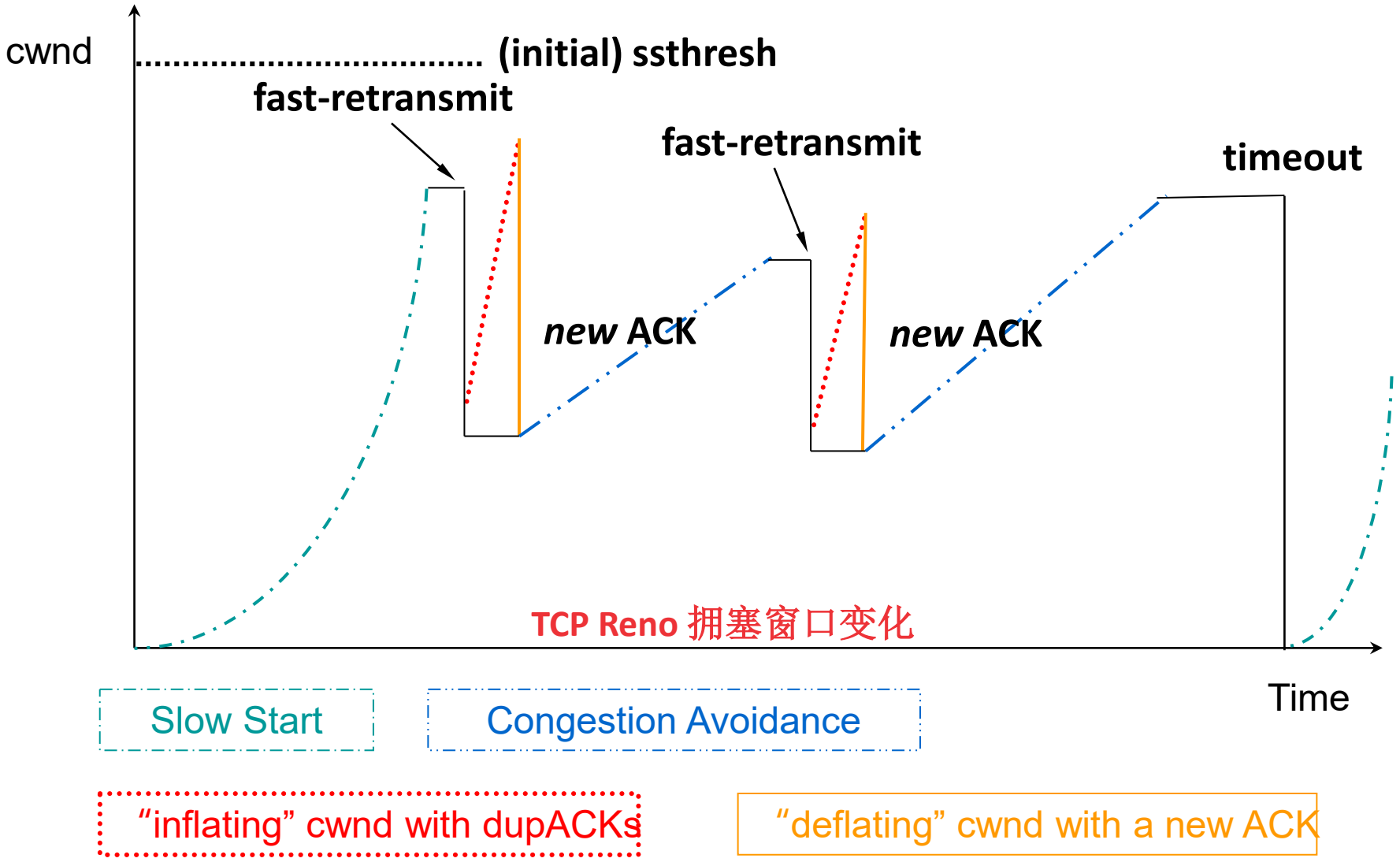
- 快速恢复阶段：1个RTT
- 快速恢复结束前，可发送w/2-1个新TCP段 + 1个快速重传的TCP段

TCP Reno总结


Λ
cwnd = 1 MSS
ssthresh = 64 KB
dupACKcount = 0
slow start
duplicate ACK
dupACKcount++
new ACK
cwnd = cwnd+MSS
dupACKcount = 0
如果允许，发送新分组
New ACK!
cwnd >= ssthresh
Λ
congestion avoidance
new ACK
cwnd = cwnd + MSS (MSS/cwnd)
dupACKcount = 0
如果允许，发送新分组
New ACK!
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
重传丢失分组
timeout
ssthresh = cwnd/2
cwnd = 1 MSS
dupACKcount = 0
重传丢失分组
duplicate ACK
dupACKcount++
New ACK!
New ACK
cwnd = ssthresh
dupACKcount = 0
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
重传丢失分组
dupACKcount == 3
ssthresh= cwnd/2
cwnd = ssthresh + 3
重传丢失分组
fast recovery
duplicate ACK
cwnd = cwnd + MSS
如果允许，发送新分组

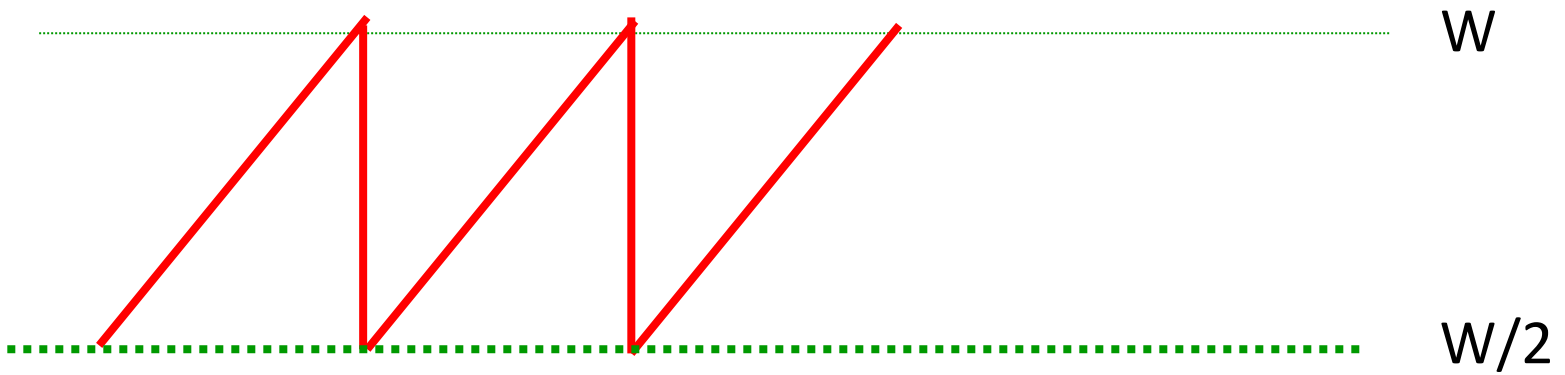
TCP Reno

- 慢启动 + 拥塞避免 + 快速重传 + 快速恢复



拓展：TCP吞吐率模型

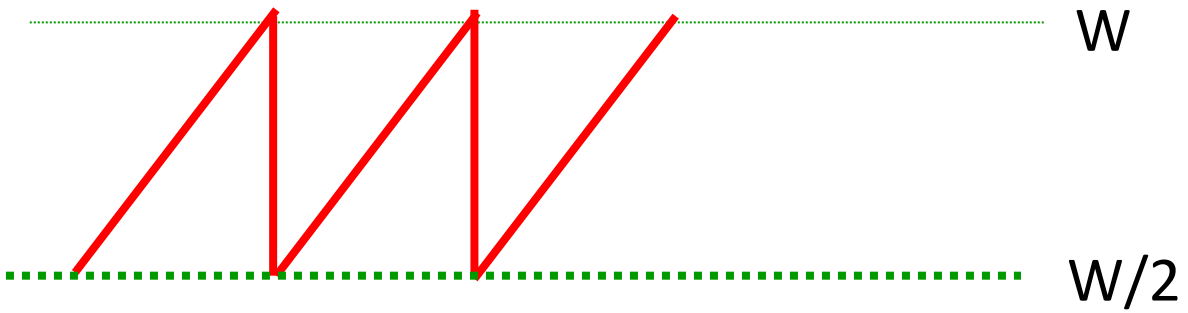
- TCP的吞吐率相关重要因素：
  - 丢失事件率(Loss event rate)：影响cwnd降低的频率
  - RTT影响窗口增加速率
  - RTO影响差错恢复阶段的性能
  - MSS 影响窗口增加速率
- 没有超时的稳定状态下TCP的行为：忽略慢启动阶段，忽略快速重传/快速恢复（1个RTT）
  - RTT固定
  - 没有delayed ACK



TCP吞吐率模型

考虑AIMD机制

- 每次拥塞窗口达到W时，1个(携带MSS数据)分组丢失
- 拥塞窗口减少到W/2分组，然后线性增加
  - 每个RTT，拥塞窗口增加一个分组
  - 下次丢失前拥塞窗口从W/2 →W
- 每个周期：
  - 时间：W/2个RTT
  - 传输的分组个数 = W/2 + W/2+1 + …+W = $3W^2/8 + 3W/4 \approx \mathbf{3W^2/8}$ 当W较大时



- 平均吞吐率 $BW = \frac{\frac{3w^2}{8} * MSS}{\frac{w}{2} * RTT} = \frac{3}{4} * MSS * \frac{W}{RTT}$
- 分组丢失率p=?
  - 每个周期传输分组数$3W^2/8$，其中有1个丢失

$p = \frac{8}{3w^2}$ ==> $W = \sqrt{\frac{8}{3p}} = \frac{4}{3} \times \sqrt{\frac{3}{2p}}$

$$BW = \frac{MSS}{RTT} * \sqrt{\frac{3}{2p}} = \frac{1.22 * MSS}{\sqrt{P} * RTT}$$

TCP Friendly

- TCP流量占整个Internet流量的大部分
- 任何新的拥塞控制必须和TCP流进行竞争
  - 不应该抢占TCP流而占有大部分链路带宽
  - 同时也应该拥有自己的公平的带宽
- 怎样来实现？
  - 采用类似于TCP的AIMD的拥塞控制机制
  - 通过TCP模型：丢失/吞吐量模型
    - 如果**吞吐量符合1/sqrt(p)行为**，则ok
    - TCP Friendly Rate Control (TFRC)

$$T = \frac{s}{R\sqrt{\frac{2p}{3}} + t_{RTO}(3\sqrt{\frac{3p}{8}})p(1+32p^2)} \quad (1)$$

This gives an upper bound on the sending rate T in bytes/sec, as a function of the packet size s, round-trip time R, steady-state loss event rate p, and the TCP retransmit timeout value t_{RTO}.

长肥网络(Long Fat Network)下的TCP拥塞控制

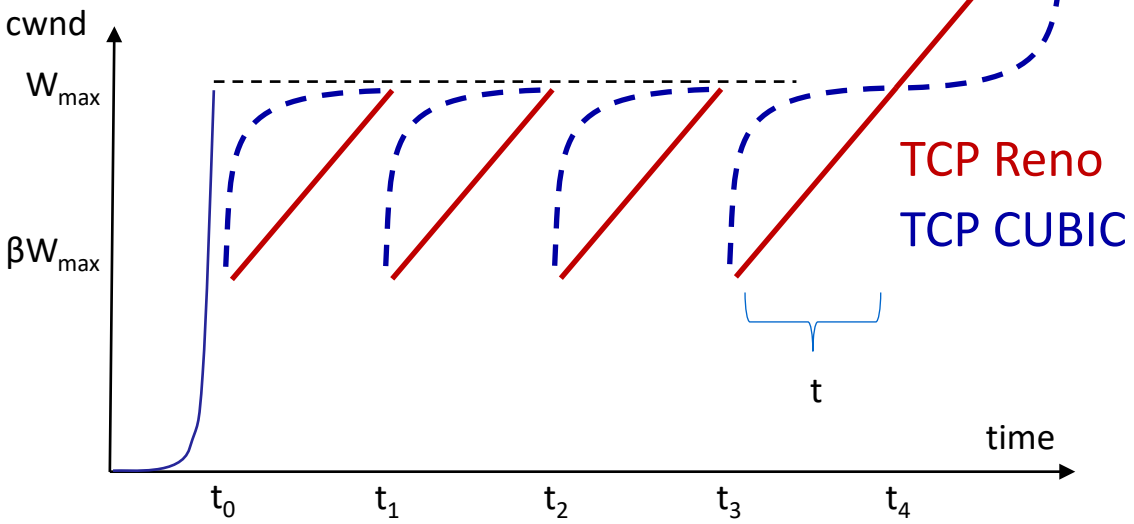
- RTT * Bandwidth **远远超过10^5 比特或12500字节**，则称为长肥网络LFN，发音elephan(t)
- 16比特的接收窗口：通过WSSCALE选项扩充为最大$(2^{16}-1)2^{14}$ =1,073,725,440字节，大约1G字节
- 32比特的顺序号窗口：避免顺序号回绕后原顺序号的分组仍然在网络中，使用时间戳选项
- 原有的拥塞控制机制(AIMD)
  - LastByteSent – LastByteAcked <= min(cwnd, rwnd)，要能够有效利用网络的带宽，需要比较大的拥塞窗口
  - 仍然采用AIMD的话，需要很长时间cwnd才能达到网络的容量
  - 基本思想：
    - 拥塞发生而倍数减少时，减少得少一些，比如 cwnd = cwnd * 7/8
    - 拥塞避免阶段：每RTT拥塞窗口增加得更快一些
- Linux系统的TCP的拥塞控制机制缺省为TCP CUBIC
- 微软操作系统下的TCP的拥塞控制机制缺省为Compound TCP → TCP CUBIC

拓展：TCP CUBIC， RFC 9438

- 类似于TCP Reno，仍然采用慢启动、**拥塞避免**、快速重传和快速恢复阶段
- 拥塞避免（线性增加阶段）阶段：
  - 起始点：拥塞事件(3个重复的ACK）发生后从结束快速恢复阶段之后，拥塞窗口从拥塞事件的 W_{max}→βW_{max}， β= 0.7， TCP Reno中β=1/2
  - 在某个时刻t，拥塞窗口满足cubic函数：$W(t) = C(t-K)^3 + W_{max}$
    - 常量C，C越大，拥塞窗口增长的速度越快，建议为0.4
    - t: 进入当前拥塞避免阶段的时间间隔
    - K: 进入当前拥塞避免阶段后拥塞窗口增加到W_{max}的时间间隔

W(0)=βW_{max}，可知

$$K = \sqrt[3]{\frac{(1-\beta)W_{max}}{C}}$$

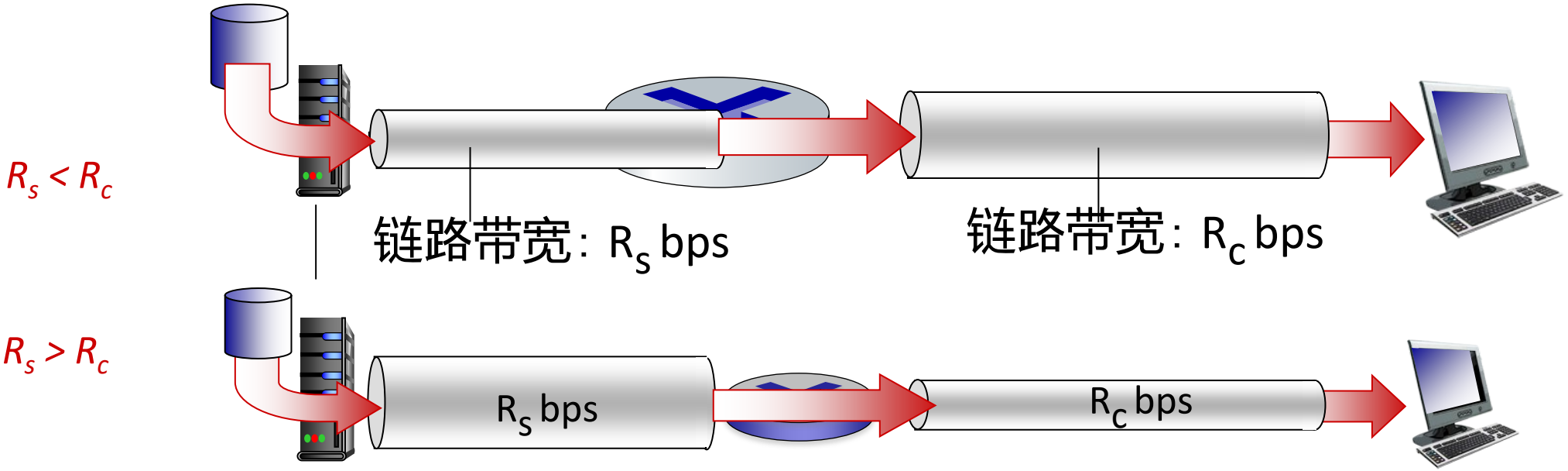


- RTT-fairness: 经过同一瓶颈链路的不同RTT的TCP流公平共享带宽
- 在什么时候调整拥塞窗口呢？具体实现是在时刻t**收到一个新的ACK**之后，计算下一个RTT时刻的拥塞窗口W(t + RTT)

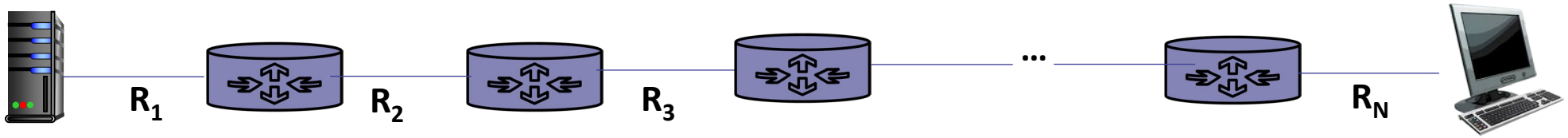
$$cwnd \mathrel{+}= \frac{w(t+RTT) - cwnd}{cwnd}$$

瓶颈链路(bottleneck link)

- Client和server的路径有两条链路，不考虑拥塞控制、处理和转发速度以及协议头部开销等因素，Client和Server之间的平均吞吐率最大为多少?



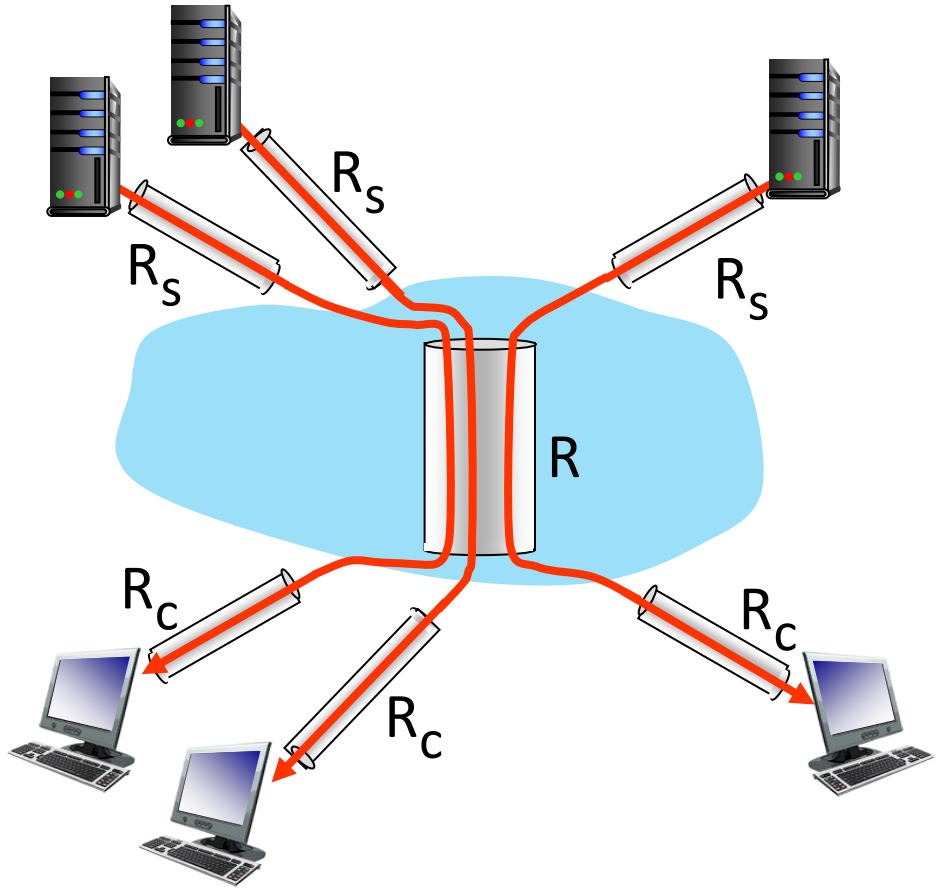
- 端到端的路径上的平均吞吐率受到瓶颈链路的制约 = $min(R_s, R_c)$



$min(R_1, R_2, R_3, ..., R_N)$

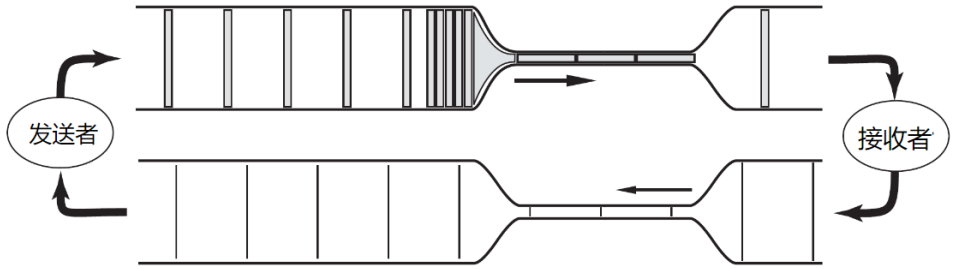
瓶颈链路

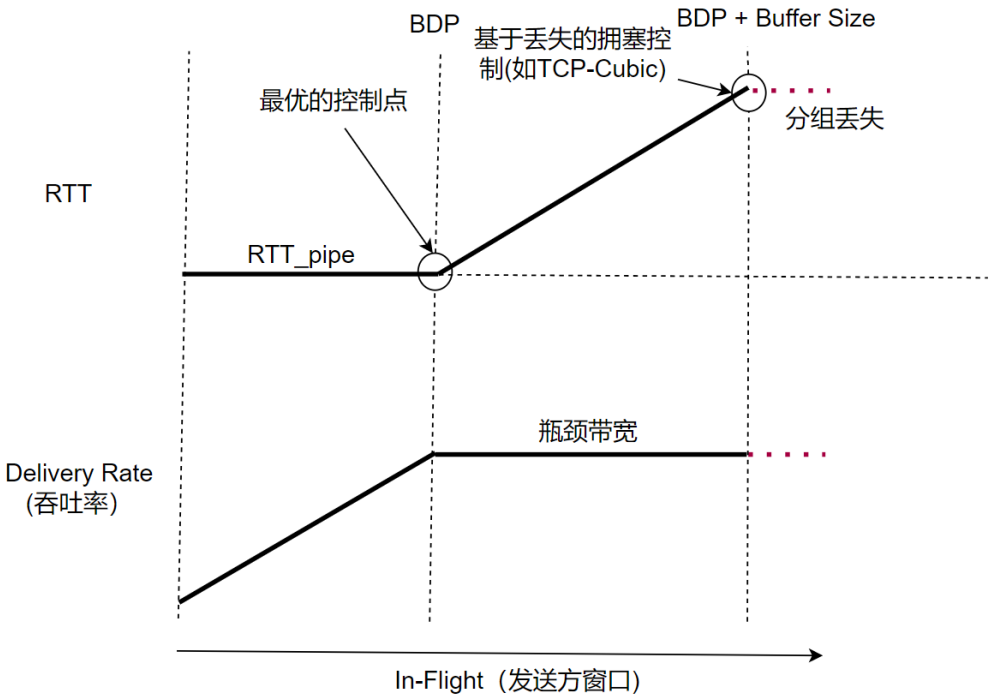
- 10个client和10个server之间建立10条连接，这些共享backbone瓶颈链路R bps
- 每条连接的平均吞吐率最大为 $min(R_c,R_s,R/10)$
- 实践中一般最后一跳链路为瓶颈链路



拓展：Bufferbloat

- 传统TCP拥塞控制（TCP-Reno/TCP-Cubic）为Loss-based，依赖于分组丢失调整发送速率（拥塞窗口）
- TCP连接的吞吐率（最大数据递交速率，delivery rate）受到瓶颈链路带宽的限制
- 瓶颈链路的缓冲区太大导致**缓冲区膨胀**
  - 缓冲区用于处理分组的突发到达，提高链路的利用率
  - 采用FIFO的缓冲区太大时，TCP发送者只有检测到丢失事件时才会减少拥塞窗口，导致分组都经历较长的延迟
- 如何应对缓冲区膨胀?
  - 减少缓冲区大小。设备厂商更愿意提供更多缓冲区的设备，更高的产品定价
  - 主动队列管理（Active Queue Management）：
    - RED 随机早丢弃：教材9.5.2，队列长度超过阈值时开始丢弃
    - CoDel(Controlled Delay)：在最近一段时间(RTT)的最小的分组逗留时间超过TARGET(5ms)时，说明是一个坏队列(standing queue)，丢弃分组
  - 端系统不采用基于丢失的拥塞控制，而是采用BBR（Bottleneck Bandwidth and Round-trip propagation time)





教材目录

3.3 数据链路协议：可靠数据传输

- 停等协议
- 滑动窗口协议：GBN和选择重传

6.2.1 TCP概述

6.2.2 TCP段格式

6.2.4 TCP可靠传输

6.2.5 TCP流量控制

6.2.3 TCP连接管理

TCP可靠数据传输

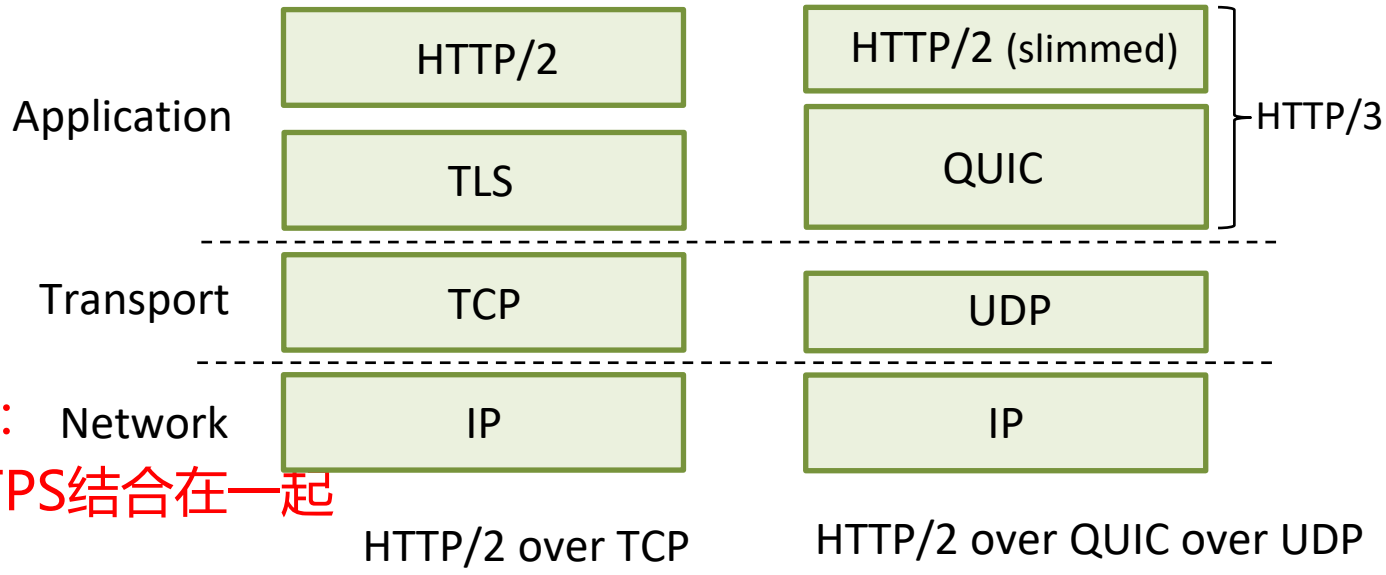
6.2.7 TCP计时器

6.2.6 拥塞控制

6.3.1 UDP

用户数据报 UDP: User Datagram Protocol

- UDP在IP层上附加了简单的多路复用功能，即允许多个应用程序通过socket接口将数据从源端主机发送给目的端主机
- UDP是一种无连接方式的、不可靠的运输协议
  - 不需要连接建立和释放
  - 没有差错控制机制，仅仅包含**可选的差错检测**机制
  - 没有流量控制、拥塞控制
  - 尽力递交服务，传输过程中可能会丢失，可能会失序，可能会延迟等
  - 支持广播和组播
- UDP协议非常简单，适合：
  - 多媒体应用：允许丢失，有速率要求
  - 单次请求/响应应用：DNS、SNMP等
- UDP应用可自己实现可靠传输机制
- QUIC(Quick UDP Internet Connections):
  - 连接建立，差错控制和拥塞控制，HTTPS结合在一起



UDP数据报(Datagram)

- 源端口和目的端口
- 长度：
  - 包括了固定长度的UDP头部和携带的用户数据
  - 长度字段冗余，= IP总长度 - IP头部长度
- 校验和：
  - (96比特的)伪头部+UDP数据报：防止错误递交
  - 采用Internet checksum算法，检验和计算采用二进制反码运算（带进位的加法，进位加到个位)，最后取其反码
  - 全1和全0在反码中对应着-0和+0，都可以表示0
  - 检验和字段为0时，表示不进行差错检测
  - 计算出的检验和为全0时，检验和字段为全1