现代计算机网络

李伟明 lwm@hust.edu.cn

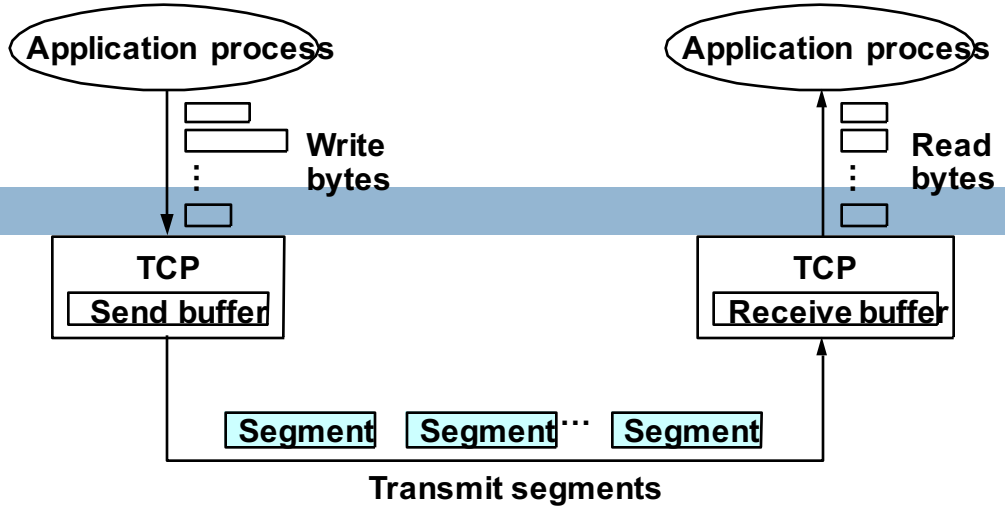
Ch.3 拥塞控制

- 3.1拥塞控制相关基础
  - 拥塞控制的基本原理
  - 拥塞控制方法分类
  - 网络模型及特点
  - 资源分配机制
  - 资源分配评价标准
- 3.2排队规则与流量整形
  - FIFO
  - 优先排队
  - 公平排队FQ
  - 加权公平排队WFQ
  - 流量整形
  - 令牌桶算法
- 3.3 TCP拥塞控制
  - 加乘式
  - 慢启动
- 3.4 TCP拥塞避免
  - DECbit
  - RED和源拥塞避免

3.3 TCP拥塞控制(闭环，源抑制)

- 源算法
  - 最广泛使用的TCP协议中的拥塞控制算法
  - 下表给出拥塞控制源算法的简单描述

拥塞控制源算法	描述
Tahoe-TCP	慢启动、拥塞避免、快速重传三算法.（早期较为普遍采用的版本）paper： congest avoid.pdf
Reno-TCP	RFC5681加上快速恢复.
NewReno-TCP	引入了部分确认和全部确认的概念.
SACK-TCP	规范了TCP中带选择的确认消息.
Vegas-TCP	采用带宽估计,缩短了慢启动阶段的时间，源拥塞避免 /net/ipv4/tcp_vegas.h， /net/ipv4/tcp_vegas.c



源端口Source Port（16 bit）								宿端口Destination Port（16 bit）
序列号Sequence Number（32 bit）								
确认号Acknowledgment Number（32 bit）								
数据偏移（Data Offset（4bit）	保留（为0）Reserved（6 bit）	URG	ACK	PSH	RST	SYN	FIN	Advertised Window（16 bit）
校验和Checksum（16 bit）								紧急指针Urgent Pointer（16 bit）
可选项Option（32 bit）								
数据Data（32 bit）								

TCP协议数据格式

- TCP基本策略：
  - 智端+傻网，端端拥塞控制，
  - 无预约,先渐增发，后观察事件反应
- TCP假设网络中R仅用FIFO排队；但也可能使用FQ规则
- 拥塞包在R中丢弃后，由端来重传解决

- TCP又叫自定时（self clocking）
  - 重传定时器
  - 坚持定时器（接收窗口为0，期性地向接收方查询，发一个字节）
  - 保活定时器（默认是2小时）
  - 2MSL（报文段最大生存时间）定时器
  - ……
- 以下描述3个TCP拥塞控制机制
  - 2.3.1 慢启动
  - 2.3.2 加法式增加/乘法式减少拥塞窗口
  - 2.3.3 快速重发和快速恢复

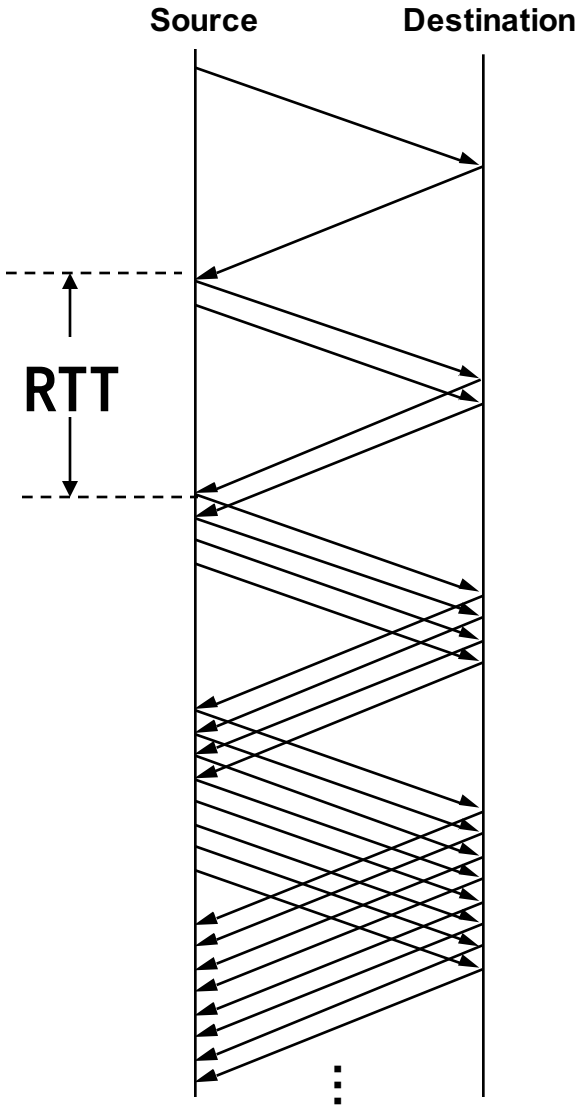
3.3.1 慢启动和拥塞避免

- 线性增加发送量
  - 对网络接近满载来说是一个正确的方法；
  - 但对从零开始起步的连接来说时间又太长；

- TCP拥控第1个机制—慢启动（实际相当快）
  - Congestion Window是发送方设置的一个变量
  - 指数式增加CW，而不是线性。
  - 开始，源把CW置为1（一个包）；
  - 然后每收到一个ACK，将CW*2
  - 当该包的ACK到达后，TCP可以发2个包；
  - 当收到2个ACK后，CW*2，发4个包，故每个RTT内包数翻一倍

慢启动传输中的包

- 慢启动称为慢的原因是，该方法未出现前，源按Advertised Window可立刻发送所有数据
- 当连接建立的时候慢启动一直到丢包才结束。
- 如果发生丢包后，，源端再次进入慢启动；当超过一个发送方设置的阈值：**ssthresh（slow start threshold）**，CW会变为线性增加，这其实是一种拥塞避免



拥塞窗口和可发送窗口

- Congestion Window （CW）
  - 源端TCP为每个连接维护的一个状态变量；
  - 限制在给定时间内容许传输的数据量，慢启动的最大值也受到CongestionWindow影响
- 容许的最大未确认字节数就是现在的CongestionWindow和AdvertisedWindow小的那个
  - MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
- 可以发送的窗口大小，为可能新增的未确认字节数：
  - EffectiveWindow = MaxWindow - (LastByte - LastByteAcked)

但发生拥塞的时候一般CW < EffectiveWindow

TCP怎样学到CW？

- 注意：
  - 不是靠连接的接收方来发送，没有谁发送CW值到发送方，所以是隐式反馈。

- TCP源慢启动之后：
  - 如果超过**ssthresh**，可能会导致当拥塞程度上升就线性CW
  - 当拥塞程度下降（收到ACK）时，就增加CW
  - 上述二者合在一起就叫：加法式增加/乘法式减少 AIMD

怎样感知是否拥塞？

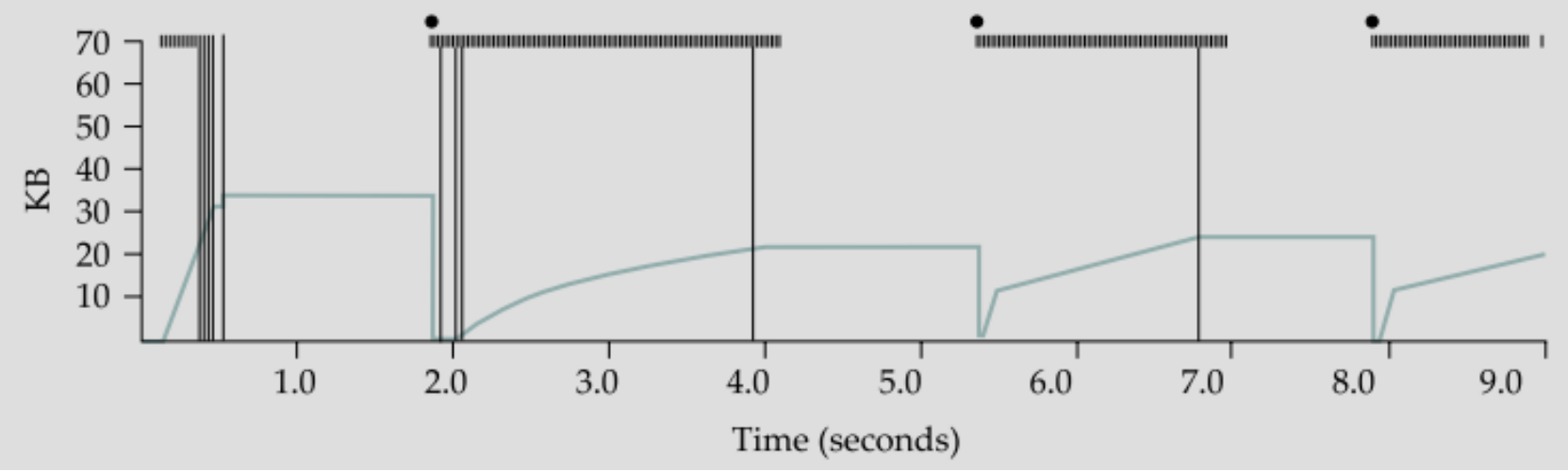
- 观察主要原因：
  - 包未交付，并导致duplicated ACK或超时，判定为拥塞丢包！
  - 由于传输错误丢包是很少的！
- TCP把收到1）duplicated ACK（对已经确认的字节再次确认），2）超时两种现象解释为拥塞的信号
  - Tahoe TCP不区分这两种情况
    - sshthresh = cwnd /2
    - cwnd 重置为 1
    - 进入慢启动过程
    - 这就是乘法减少机制的一部分
  - **Reno TCP**超时的策略不变，但是面对duplicated ACK的实现是：
    - 进入拥塞避免
    - cwnd = cwnd /2
    - ssthresh = cwnd
    - 进入快速恢复算法——Fast Recovery

从最简单的情况开始

假设TCP拥塞控制仅仅有

- 满启动
- 拥塞避免

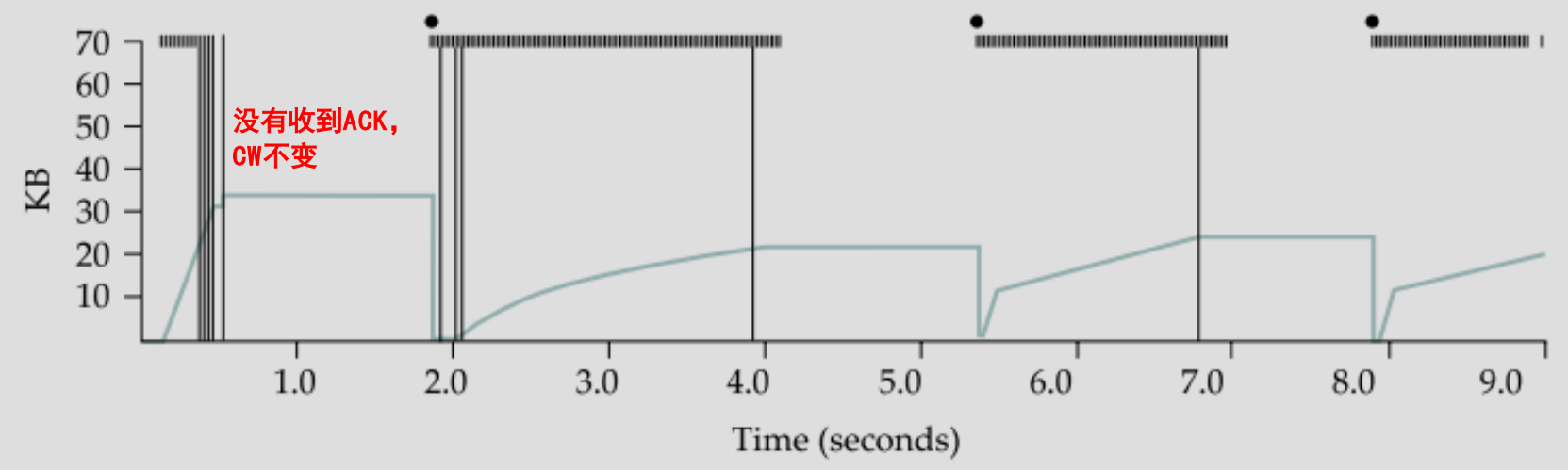
Figure 6.11 Congestion Window变化



仅仅有慢启动和拥塞避免时，TCP处理报文超时：

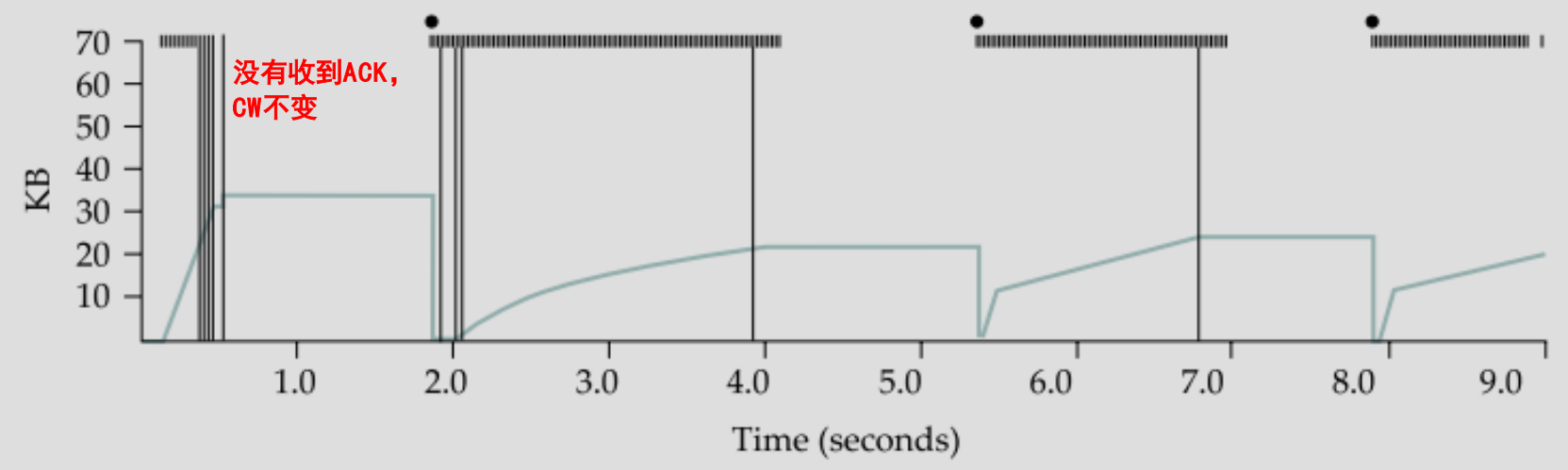
- 图中青色的线是CW的变化
- 横轴是时间，纵轴是CW的大小
- 图上端短的竖线表示发送报文，长竖线也表示发送报文，但是这个报文会丢失，这是第一次发送
- 图上端黑色圆点表示源发现报文超时

Figure 6.11 Congestion Window变化



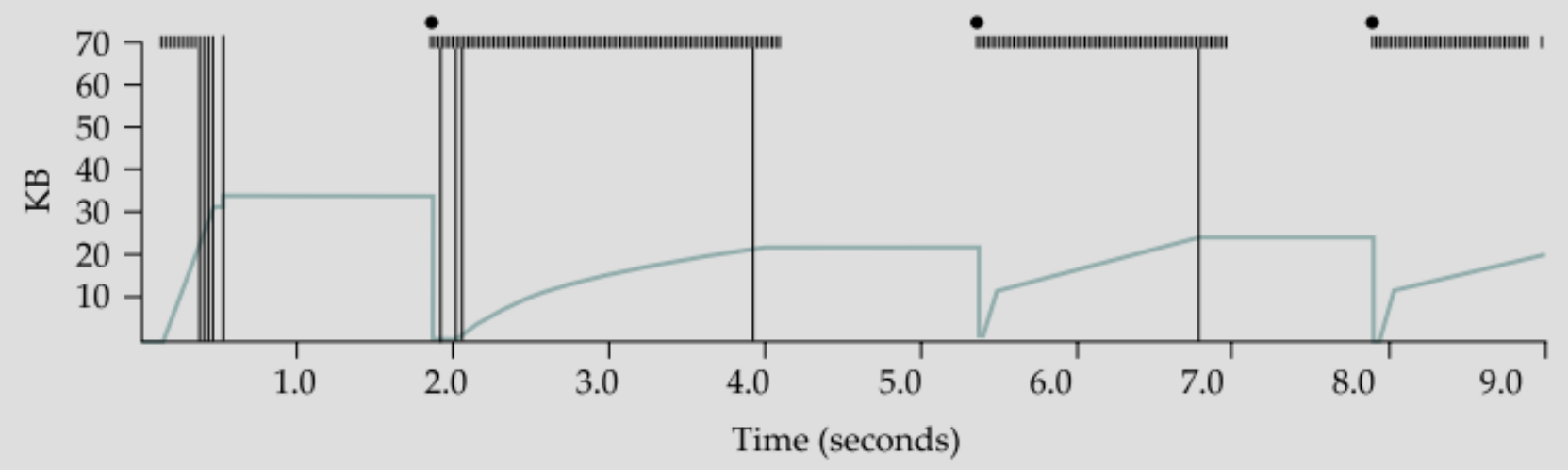
- 0.5秒发送的大量报文丢失了，所以到2.0秒附近超时，slow start threshold 降低到当前CW的一半，CW=1，慢启动开始
- 从2.0秒附近慢启动，开始重传和发送新报文，但是由于这个时候收到ACK速度较慢，所以CW慢慢爬坡
- 到4.0秒附近又收不到ACK，CW保持不变

Figure 6.11 Congestion Window变化



- 5.5秒的时候有发生了超时，CW除以2得到ssthreshold，大概从 22 to 11 KB，CW设置为1个包大小，进入慢启动;
- 这次收到ACK速度比较快，CW快速增长直到ssthreshold，然后进入拥塞避免，线性增长CW
- 第7秒又收不到ACK了，CW不变
- 第8秒附近重复慢启动

Figure 6.11 Congestion Window变化



- 为什么一开始会丢大量的报文？
- TCP在链接一开始建立是无法知道网络带宽的，所以尽力发送大量的报文来测试网络能力
- 这也是TCP一个连接唯一的一次机会

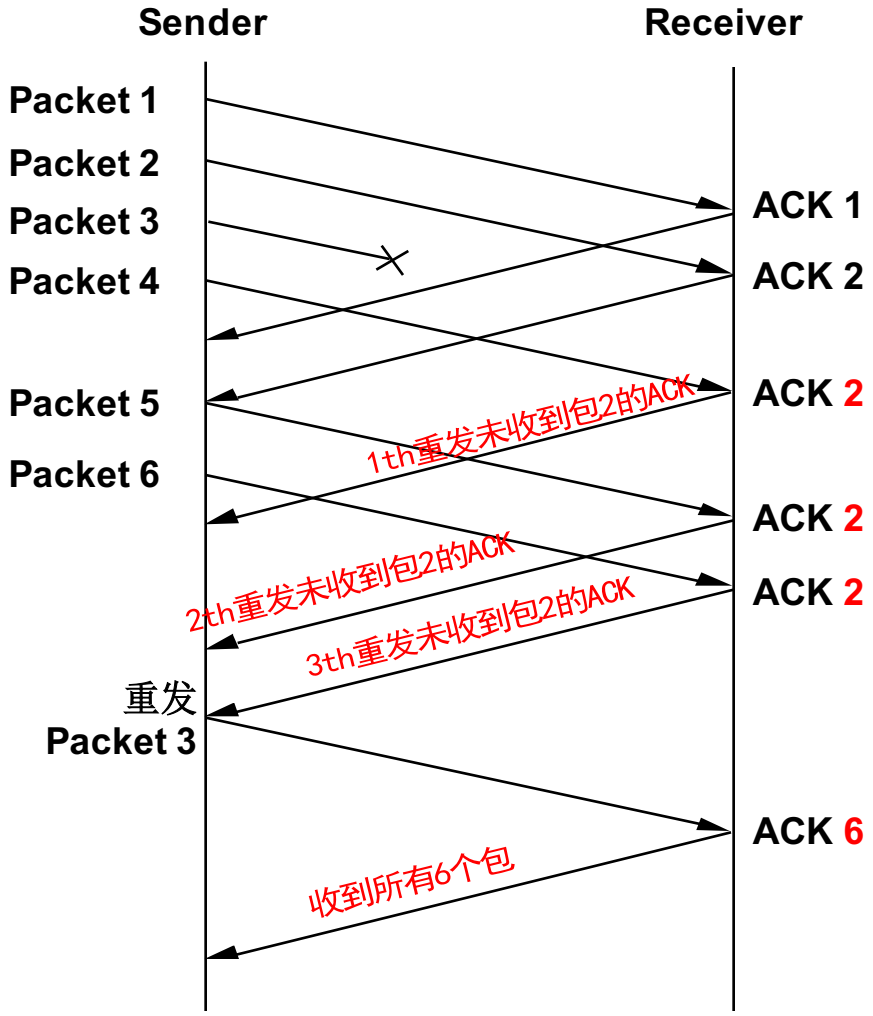
3.3.2 快速重传和快速恢复

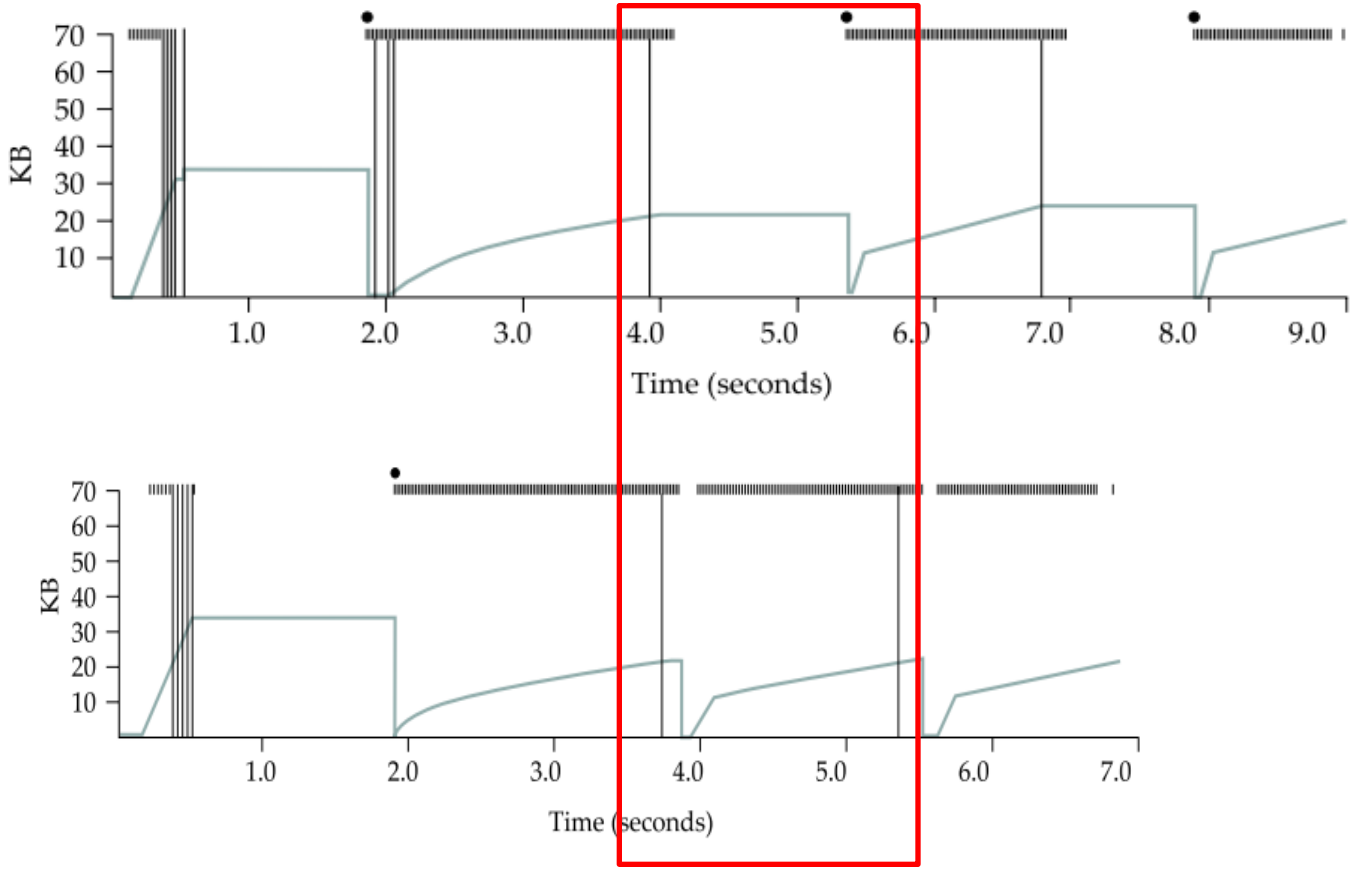
Tahoe引入快速重传，Reno引入快速恢复

- 每次接收方收一个数据包
  - 接收方响应一个ACK；（早先规定，后来优化）
  - 当包到达失序,TCP因为较早的数据还没有到达,也不能应答含有这些包的数据;
  - TCP发送它上次发送了的那个相同的ACK
  - 相同ACK的第二次发送称着Duplicate ACK,
  - 当发送方看到重复ACK,知道对方收到了一个失序包，这说明这个ACK序号后面的包丢失了
- 实际情况中
  - 发送方TCP等到3个Duplicate ACK后就重发丢失的包；即快速重传，比等待报文超时要快

重复ACK触发快速重传

- 目方收到包1和包2，但包3在网络中丢失
- 当包4到达时，目方重发包2的ACK（1th重发未收到信息）
- 当包5，包6到达时，目方仍重发包2的ACK （2th重发未收到信息）
- 当发方收到包2的第3个ACK后，它（3th重发未收到信息）立刻重发包3
- 当重传包3到达目方后，收方返回一个含有包3、6的累计ACK，





上图教课书图6.11没有快速重传，下图6.13有快速重传，明显可以看出第4秒后反应速度差异。

- 为什么快速重传是合理的？因为既然ACK报文可以通过网络，证明网络是通畅的，应该加快发送速度（resulting in roughly a 20% improvement）
- 由于快速重传第5.5秒附近的超时也消除了，因为重传报文可以导致接收方发送的ACK报文，去掉超时。

快速恢复

当快速重传同时快速恢复（Duplicated ACK）

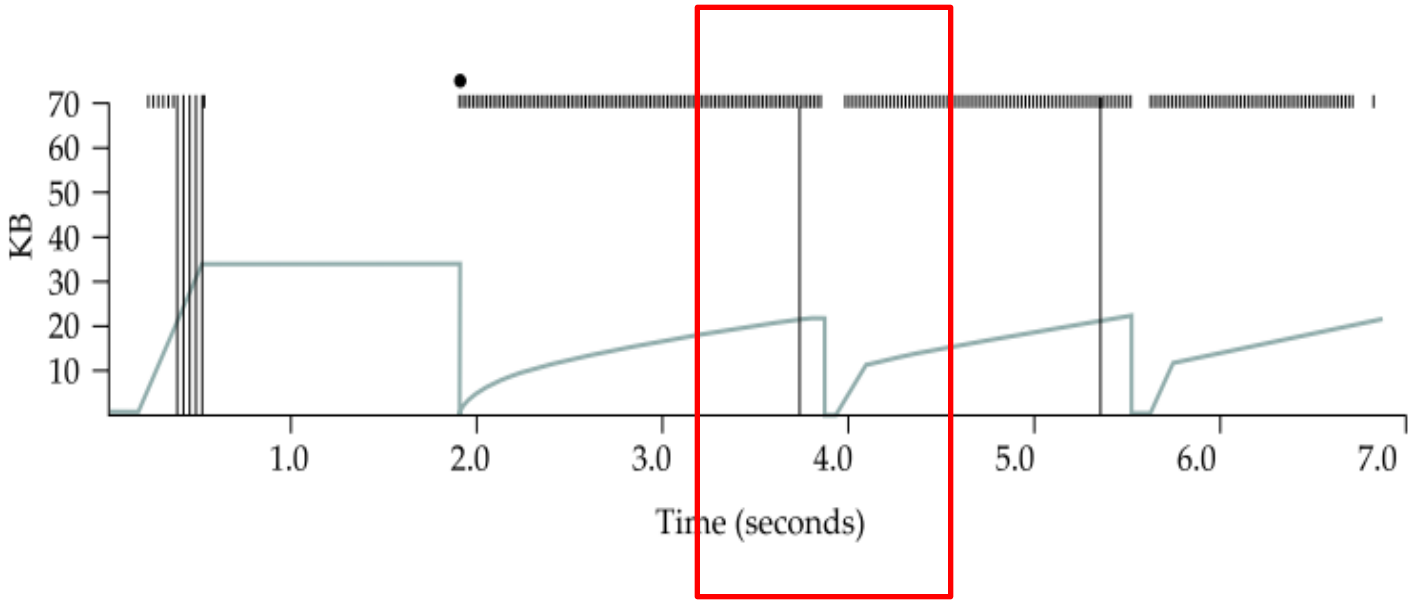
- Fast Recovery算法如下：
  - 1.把ssthresh设置为cwnd的一半
  - 2.把cwnd再设置为等于ssthresh的值，不再从1开始(具体实现有些为ssthresh+3，因为三个重复的ACK，表示有三个报文离开网络了，可以新增三个)
  - 3.重新进入拥塞避免阶段，再收到重复ACK，拥塞窗口+1（不会加剧拥塞，因为有ACK报文，说明可以增加发送）
  - 4.收到新的报文ACK时候，ssthresh变为第一步的大小（说明所有丢失的报文都收到了，可以快速恢复）

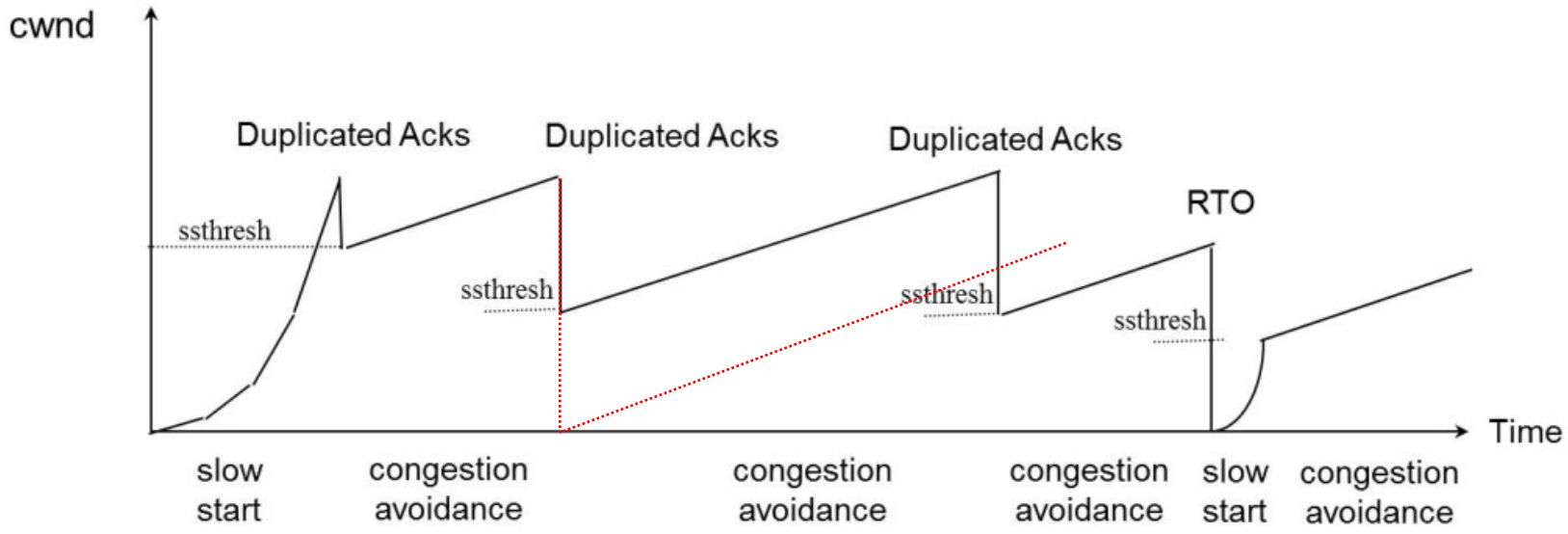
实际慢启动

- 仅在连接开始和超时发生时,
- 所有其它时间,CW是纯线性增加/乘式减少模式

快速恢复

下图中，如果是快速恢复，不会出现3.8秒左右的慢启动了





- 所以Reno TCP中慢启动只能发生在开始和超时阶段。（红色虚线表示优化前）
- 图中最左是开始阶段
- 图中最右部分，是发生了超时，所以需要慢启动，这时候实际上减半的是ssthresh；congestion windows是从ssthresh开始逐步增加

3.4 拥塞避免机制

和TCP拥塞控制中的拥塞避免是两个概念

- TCP的策略
  - 是一旦发生拥塞就来控制拥塞，而不是在第一个地方避免拥塞
  - TCP通过不断增加负载来影响网络的效果，从而找出拥塞发生点，然后从这一点后退。
  - TCP需要创造丢失来发现该连接的可用带宽
- 拥塞避免策略（跟之前的拥塞避免不同）
  - 拥塞将要发生时进行预测，在包将被丢弃之前减少主机发送数据的速率
  - 存在不同的拥塞避免机制：
    1）DECbit类机制
    2）路由器主动队列管理

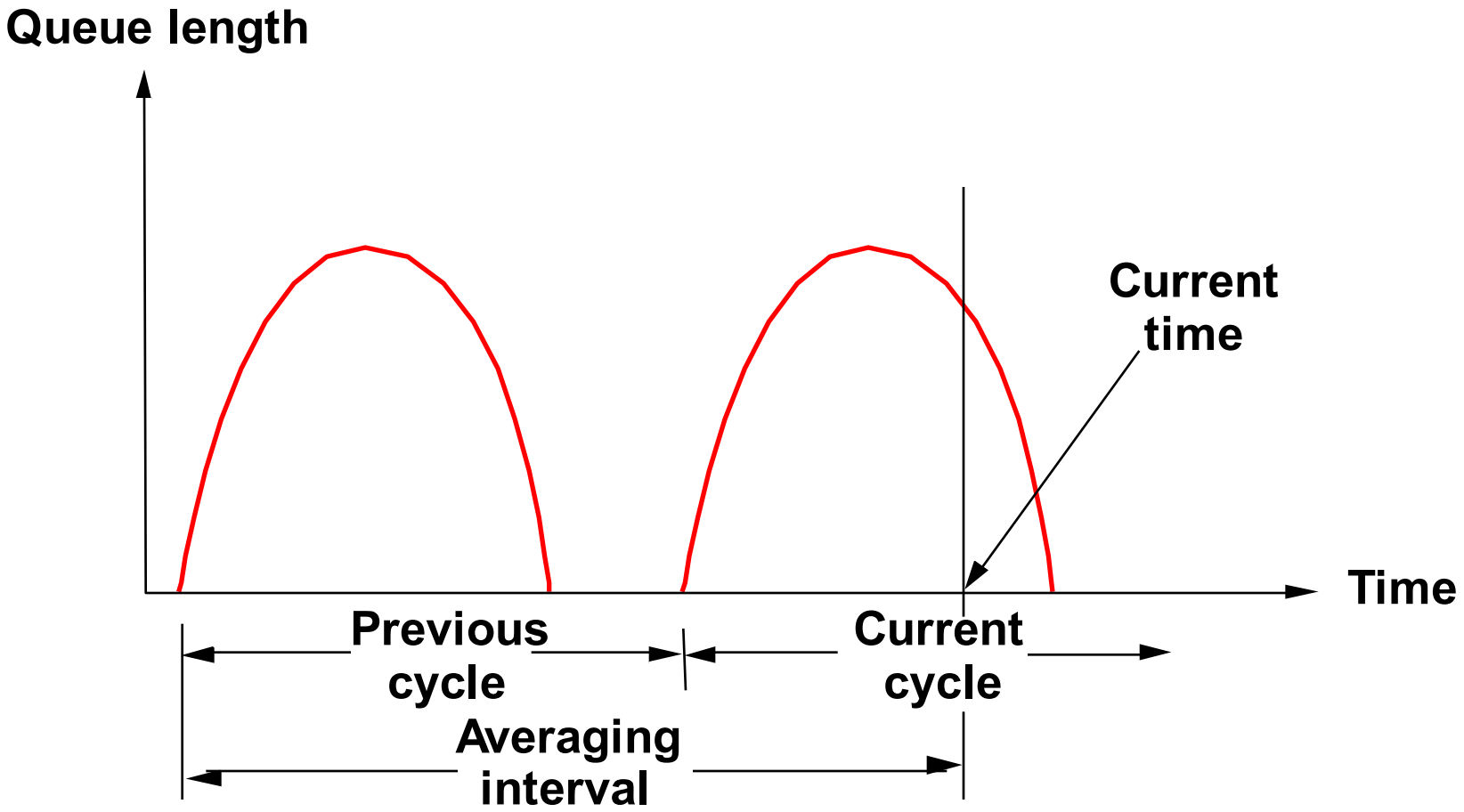
3.4.1 DECbit

- 第一种拥塞避免机制
  - DEC公司开发，故称 DECbit
  - Digital Network Architecture 上使用，也能用在TCP/IP上
- 基本思想：
  - 把拥塞任务更均匀分担在路由器和端节点之间
  - 每个路由器监视它正在经历的负载，在拥塞将要发生时明确地通告端节点
  - 通过设置流过路由器包中的二进制拥塞位来实现,目端把这拥塞位拷贝进ACK报文并返送回源端
  - 源端调整自己的发送速率以避免拥塞，设置标志位告诉接收方，已经处理了拥塞

拥塞位的设置

- 当包到达
  - 且平均队列长度大于或等于1时，路由器就设置包头中的拥塞位
  - 测量平均队列长度的时间间隔是
    - 最近的忙期+空闲期+当前忙期（R传输时为忙，不传输时为闲）
  - 路由器计算曲线下面积（下一张ppt中的图）并除以时间间隔得到平均队列长度
- 用队列长度1作为设置拥塞位的触发器
  - 是在有效排队(维持高吞吐率)和增加闲期(较低延迟)之间的折中,即队长1是网络能力（吞吐率/延迟）函数的优化

路由器中平均队长的计算



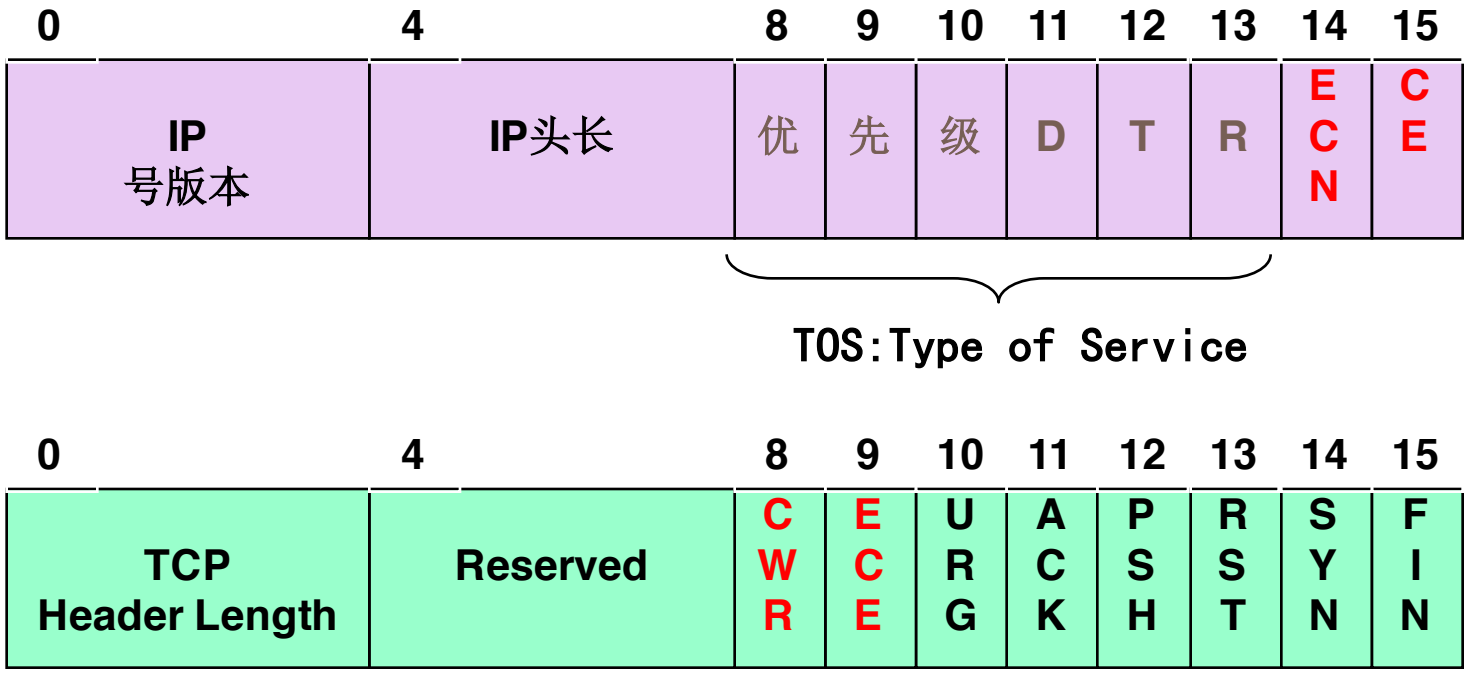
另一半：主机的工作

- 源主机的工作
  - 记录有多少包导致其在路由器中记下拥塞位
  - 维护一个拥塞窗口，并观察最近等价窗口值导致设置拥塞的比例，
    - 若比例<50%，源增加拥塞窗口1个包
    - 若比例>=50%，源减拥塞窗口至原值0.875倍
- 这符合累加/倍减机制

3.4.2 ECN显式拥塞控制

- ECN:Explicit Congestion Notification，IETF RFC 2481
  - 标记包而不是丢弃包
  - 减少超时长度和重传次数
- 一种拥塞通知方式；格式分配如下
  - IP头中设置 2 bits 的ECN域，TOS字段中的第6、7位分别定义为ECT 和CE（V4的TOS中的前6位是区分服务）
    - ECN Capable Transport(ECT)：由源主机对所有包设定，以指明ECN 能力
    - Congestion Experienced(CE)：由路由器设置作为拥塞的标记（代替丢弃）
  - TCP头中设置2 bits ，ECE 和 CWR标志位
    - ECE：Echo Congestion Experienced ,一旦目的主机收到CE,在所有包上设置ECE，直到收到源主机的CWR
    - CWR：Congestion Window Reduced，由源主机设置，以指明它收到了ECE并减少了窗口大小

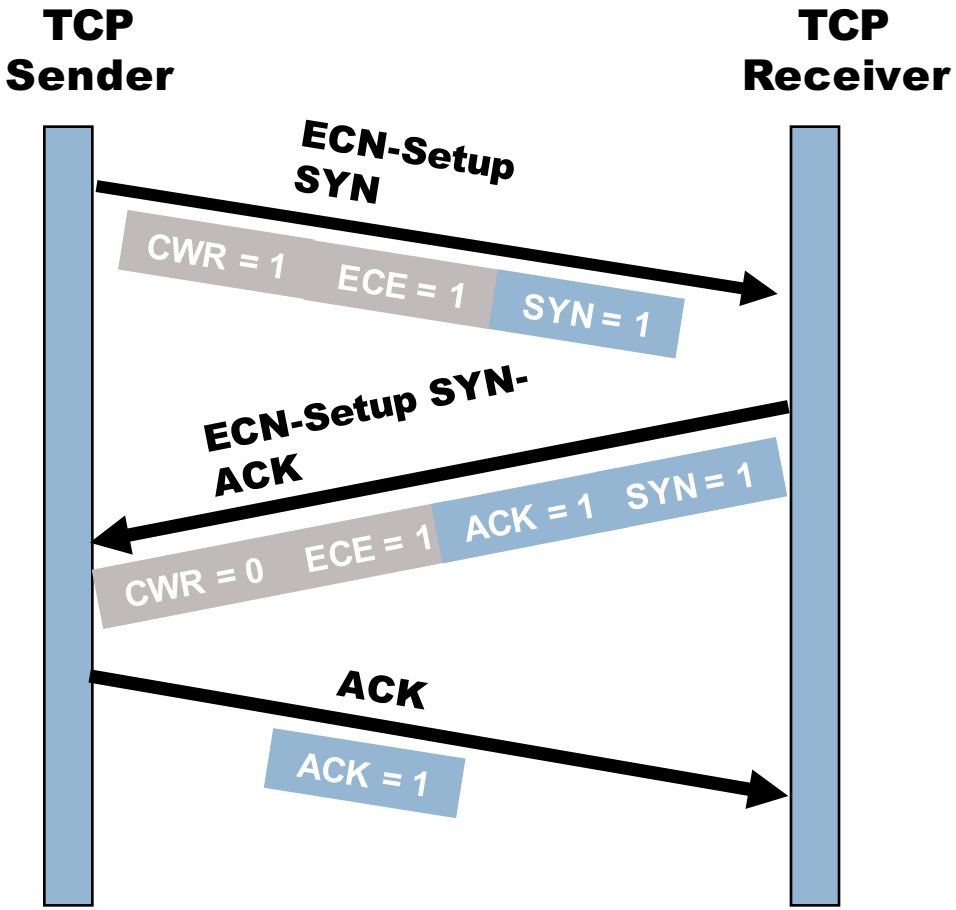
ECN 的表达



- 需要源与网络合作
  - 源主机用ECN向路由器指明开启；
  - 目的主机用ECE通知源；源主机同丢包一样反应，并置CWR

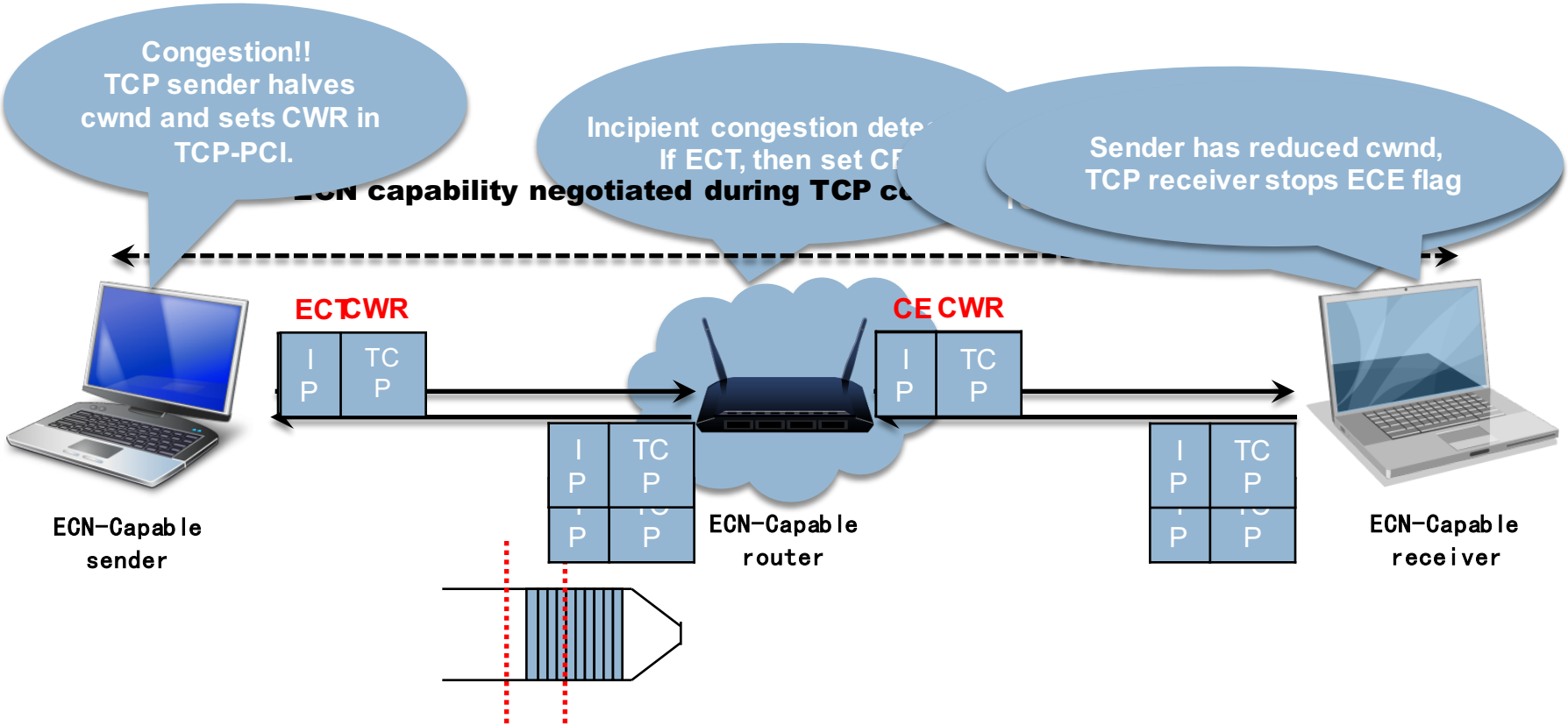
ECN 的协商

- 这种端到端的机制需要三次握手的时候协商



Example

ECT	ECN Capable Transport
CE	Congestion Experienced
CWR	Congestion Window Reduced
ECE	ECN-Echo

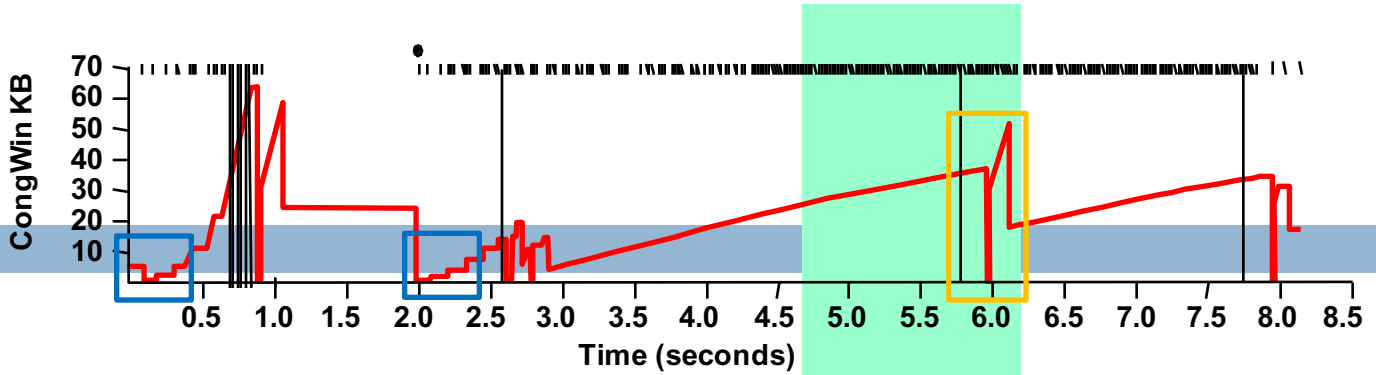


3.4.3 基于源的拥塞避免

- 丢包发生前,源用不同算法检测早期拥塞现象
  - 算法1:源观察到随R中包队列增大,后继包的RTT都会增加,如每隔2个RTT,检测当前RTT是否大于迄今所测RTT的最大最小的平均值,若大则把拥塞窗口减少1/8
  - 算法2:是否改变当前拥塞窗口根据RTT和窗口2个因素的变化而决定,每隔2个RTT,计算(CurrentWindow-OldWindow)* (CurrenRTT-OldRTT),若＞0 则窗口减少1/8,若≤0 则窗口加1最大包长,每次调整使其围绕最佳点震动
  - 算法3:拥塞前发送率接近平缓.每个RTT内将窗口加1个包,同时将获得的吞吐量与加1个包前的吞吐量比较:若差小于只有1个包传输时所得吞吐量的1/2,吞吐量增加不明显，说明有可能发生拥塞，则窗口减1.通过计算未发完字节数/RTT得吞吐量

算法4:TCP Vegas

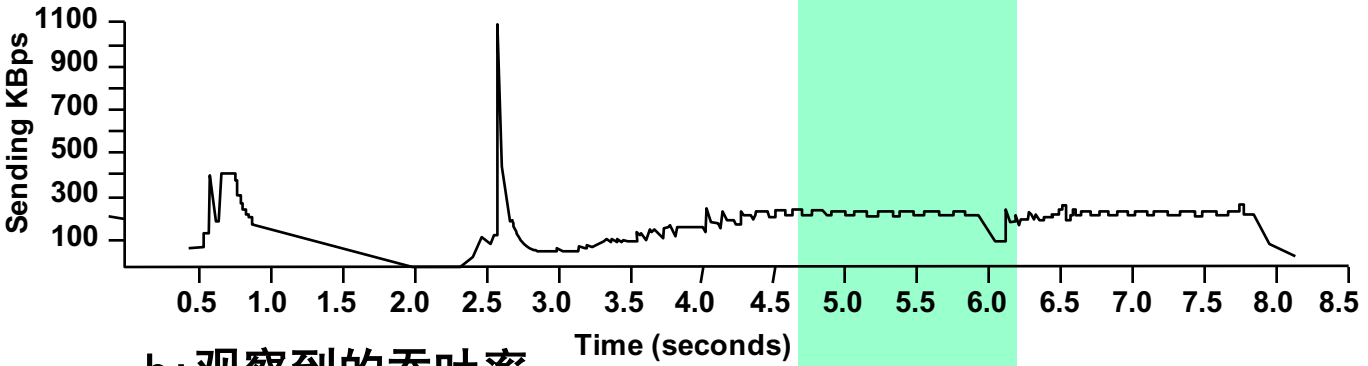
- 类似算法3
  - 查看吞吐率或发送率的变化,
  - 不同的是,它将测量的吞吐量变化率与理想吞吐量变化率比较
- 在下张PPT图中第2.0--4.5秒之间(绿色部分):
  - 拥塞窗口的增加,希望吞吐率也随之增加(a),
  - 然吞吐率却保持平缓(b),这是因吞吐率不能超过可获得的带宽,
  - 窗口的增加只会导致包在瓶颈R缓冲空间占用(c)



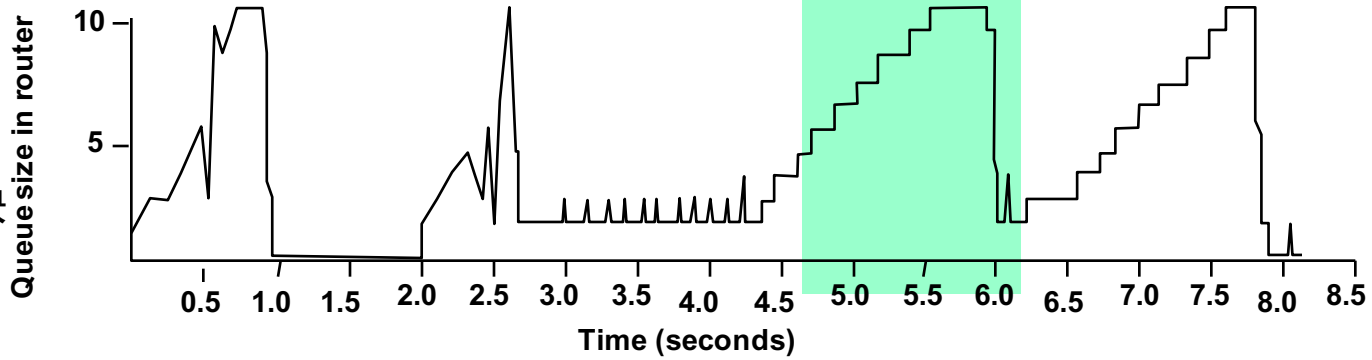
a：拥塞窗口

图6.18没有采用Veges算法的TCP拥塞窗口观察，问题在于拥塞窗口增加，吞吐率没有

- 拥塞窗口与观察到的吞吐率(3图同步)
- 有色线=拥塞窗口
- 上方黑点=超时
- 散列符号=每个包被发送的时间
- 竖条=丢包/最终将被传送的包的第一次传送时刻
- 注意，图中除了超时（蓝框）慢启动，重复ACK是快速恢复（橘色）。



b：观察到的吞吐率



c：R上占用的缓冲区

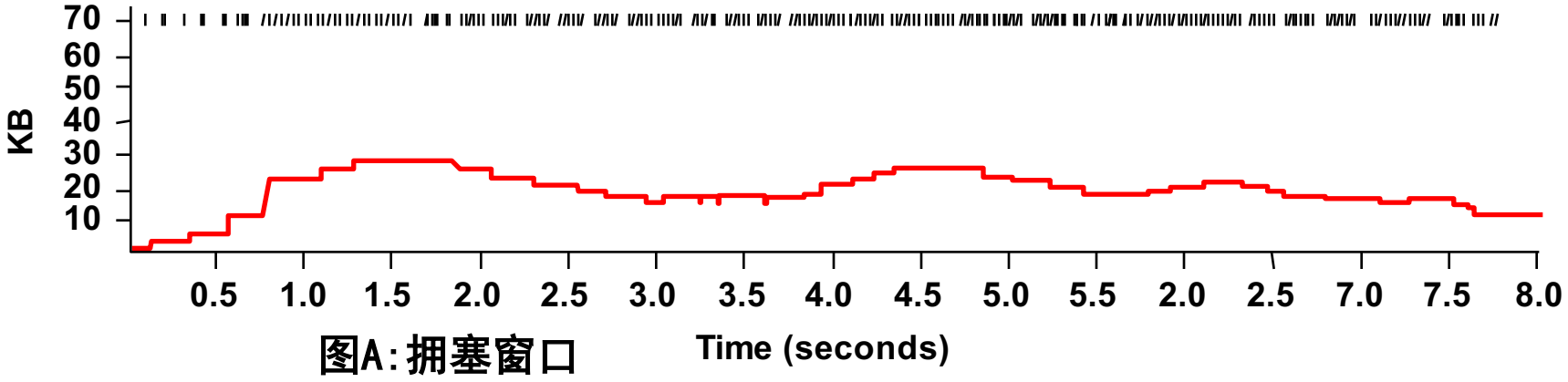
Vegas 算法

- Step1
  - 定义BaseRTT为给定流无拥塞时包的RTT值，
  - 实为测得所有往返时间中的最小值，通常是该连接发送的第一包的RTT，希望吞吐率为
  - ExpectedRate=CongestionWindow/BaseRTT
  - 假设CongestionWindow等于可以发送的字节数
- Step2
  - 计算当前发送率ActualRate, 记录每个包的发送时间和从发送到确认间包的字节数,
  - 收到确认后计算RTT,相除得发送率,每个RTT计算一次

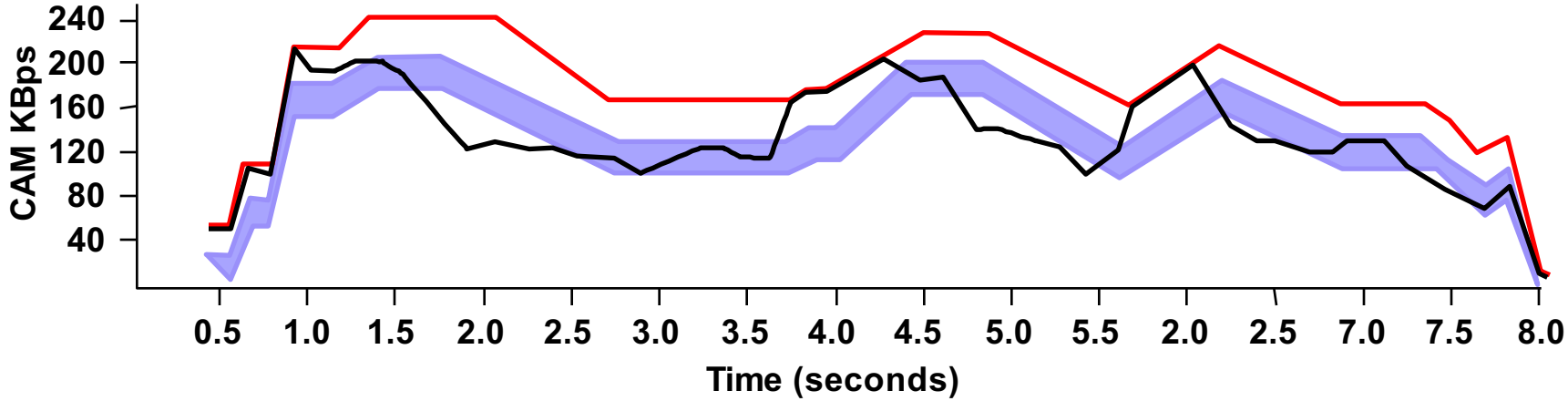
用差值调整拥塞窗口

- Step3
  - 比较ActualRate和ExpectedRate,并调整窗口,
    - Diff = ExpectedRate - ActualRate(按定义应≥0),并定义阈值α＜β,大致对应网络中太少和太多数据情况
    - 当Diff ＜ α ,则TCP Vegas 在下一个RTT中线性增加拥塞窗口;
    - 当Diff ＞ β,则在下一个RTT中线性减少拥塞窗口
    - 当α ＜ Diff ＜β TCP Vegas 保持拥塞窗口值不变
- 可以看出
  - 实际吞吐率和希望吞吐率差值越大,表明拥塞越大,故使CW窗口减少;
  - 而当二者太接近时,该连接可能不能充分利用带宽, 使CW窗口增加

TCP Vegas拥塞避免机制的过程

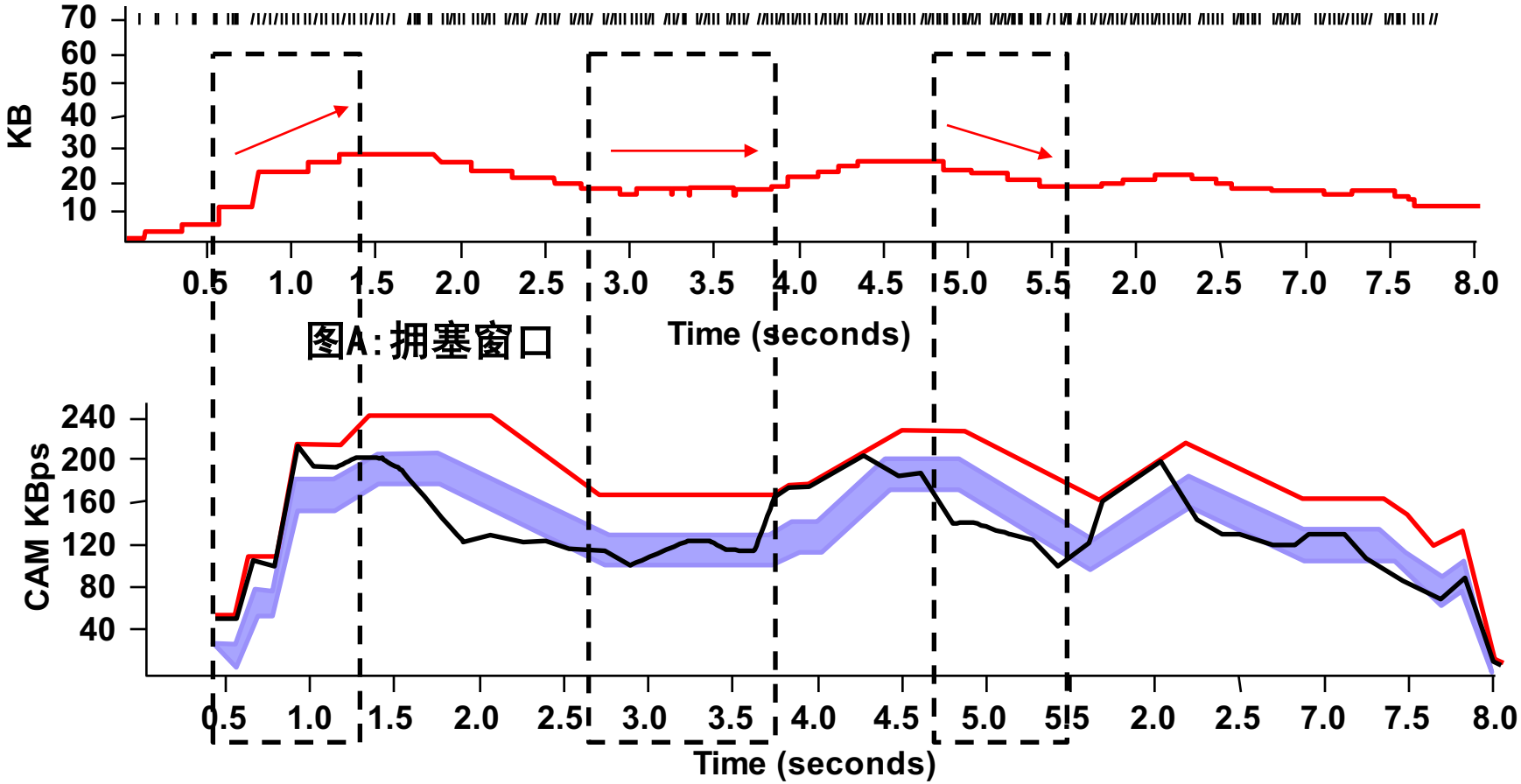


图A:拥塞窗口



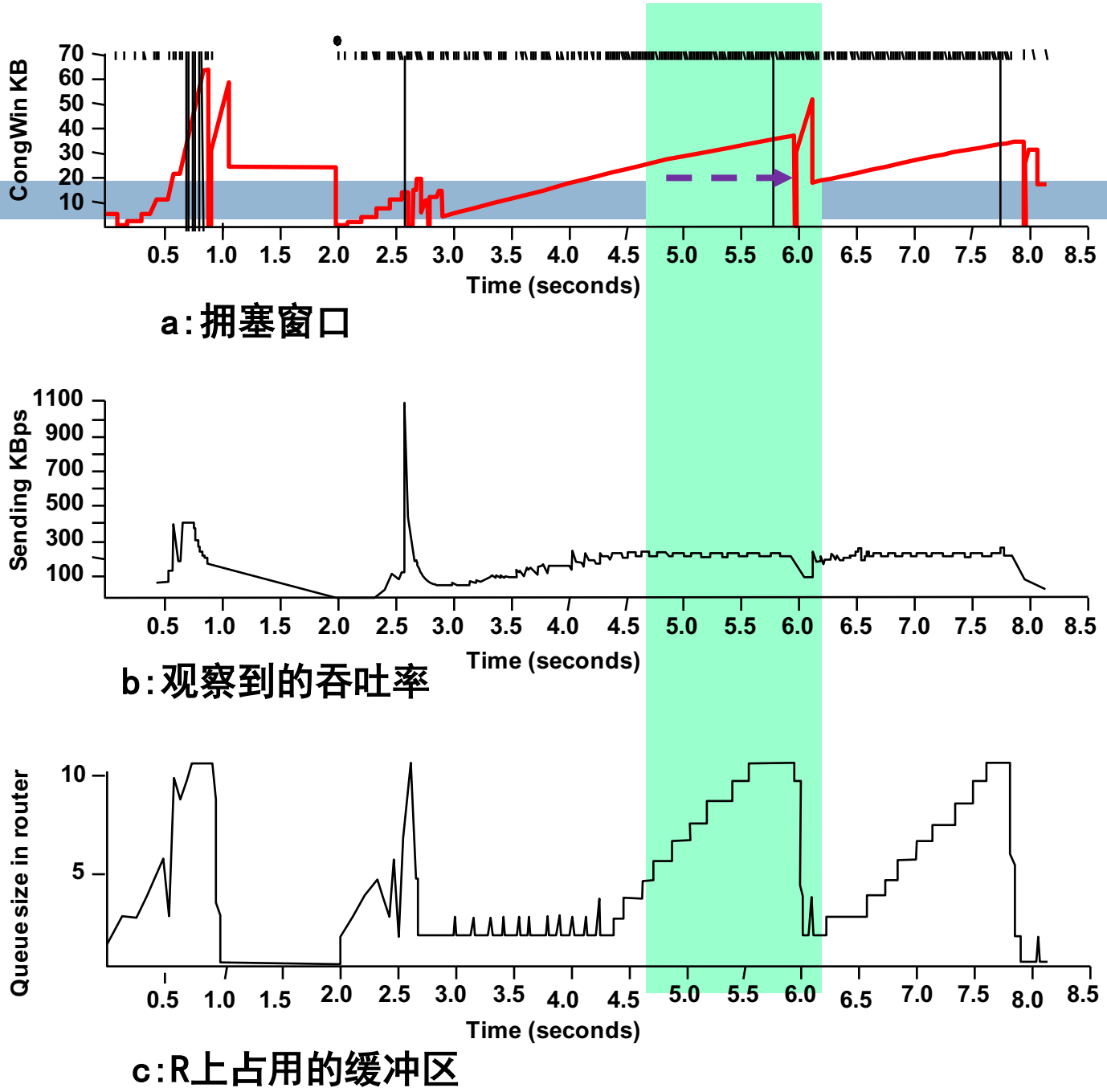
期望(红色)和实际(黑色)吞吐量,紫色阴影区是实际吞吐量应该在的α,β阈值中间的区域
紫色区域上沿是ExpectedRate-α，下沿是ExpectedRate-β，一开始期望和实际吞吐相等

TCP Vegas拥塞避免机制的过程



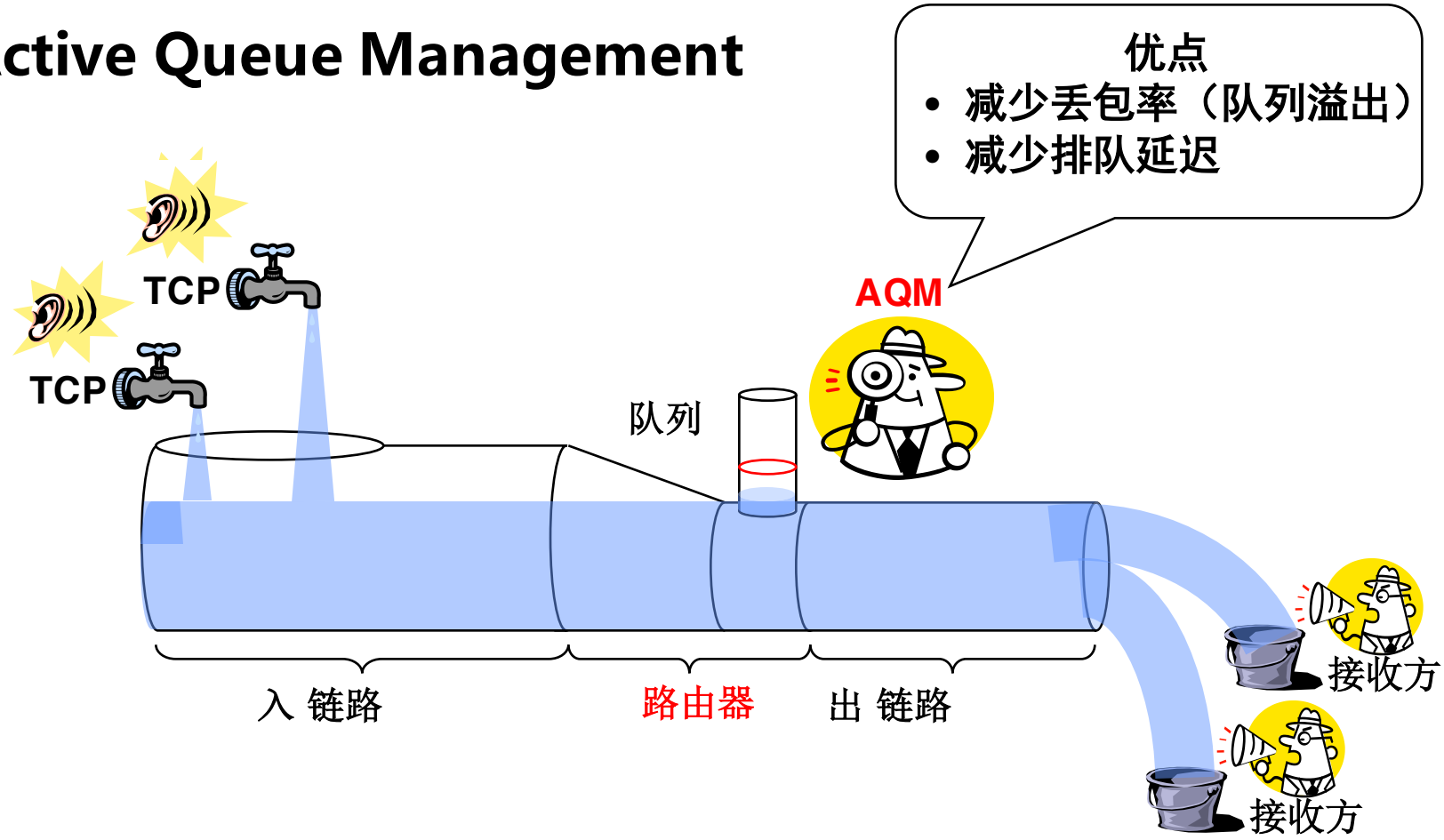
黑色的实际吞吐量要在紫色区域中，如果超过紫色区域上沿，拥塞窗口线性增加；低于紫色区域下沿拥塞窗口线性减少；在紫色区域中，保持不变。

图6.18中，注意紫色箭头，一旦发现拥塞窗口增加而吞吐率没有增加，Vegas算法会减少拥塞窗口。所以不会出现这种将额外流量发送到网上的错误做法



3.5 路由器主动队列管理AQM

Active Queue Management



为什么要 AQM？

和FIFO区别？

- AQM is the policy of dropping packets
- FIFO是排队方法，一般和tail drop绑定，当然也可以和AQM 绑定
- Drop-tail have a tendency to penalise bursty flows, and to cause global synchronisation between flows. By dropping packets probabilistically, AQM disciplines typically avoid both of these issues.

Floyd, Sally; Jacobson, Van (August 1993). "Random Early Detection (RED) gateways for Congestion Avoidance". IEEE/ACM Transactions on Networking.

为什么要 AQM？

- 控制平均队长
- 吸收没有丢弃的突发流包
- 防止非TCP友好流的突发连接行为
- 避免TCP的全局同步现象
- 减少TCP的超时次数
- 惩罚行为不端的流

基于队列的AQM

- RED Random Early Detection [Floyd and Jacobson, 1993]
  - 丢包概率与平均队列长度（缓冲区占用率）成比例
- SRED Stabilized RED [Ott et al., 1999]
  - 丢包概率与瞬时缓冲区占用率和活动流的估计数成比例.
- FRED Flow RED [Lin and Morris, 1997]
  - 使每个流的丢包率与其平均队长和瞬时缓冲区占用率成比例
- BLUE [Feng et al., 2001]
  - 用丢包事件和链路空闲事件来管理拥塞。
  - 当丢包率增加时增加丢包率；当链路空闲时减少丢包率。

基于负载的AQM

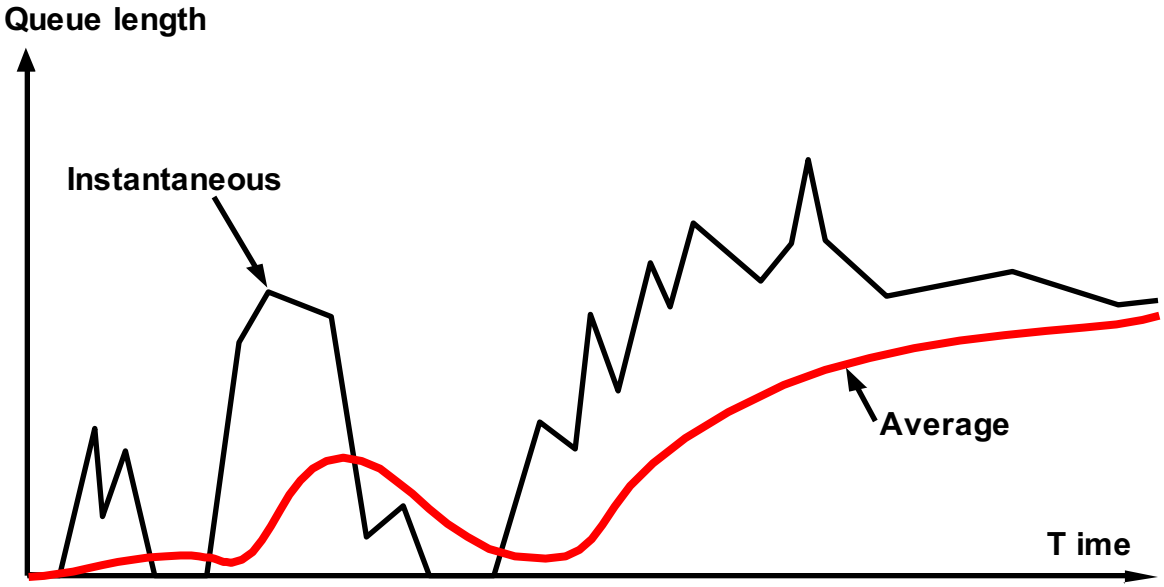
- REM Random Exponential Marking [Athuraliya et al., 1999]
  - 通过输入和输出之间速率差及路由器中当前缓冲区占用率来计算拥塞的尺度
  - 源的发送速率与拥塞尺度成反比，于是源达到全局优化平衡
- AVQ Adaptive Virtual Queue [Kunniyur and Srikant, 2001]
  - 维持一个虚拟队列.当其队列长度溢出时，实际队列中的包就标记或丢弃
  - 修改虚拟容量使所有流达到希望的链路利用率
- PI (Proportional Integral) controller[Hollot et al., 2001]
  - 队列长度的变化决定丢包率

随机早检测-RED

- Random Early Detecation
  - 与DECbit相似，都是在每个Router上监视自己队列长度
- Early
  - R在其缓冲被完全填满前就丢少量包，使源方减慢发率
- RED与DECbit的不同
  - RED不明确发送拥塞通知到源，而是通过丢弃一个包来隐含已发生拥塞。源会被随后的超时或重复ACK所提示，这可与TCP配合使用
  - 决定何时丢弃及丢弃哪个包的细节上不同，对FIFO，不是等队列完全排满后将每个刚到达包丢弃，而是当队列超过某阈值时按某概率丢弃到达包

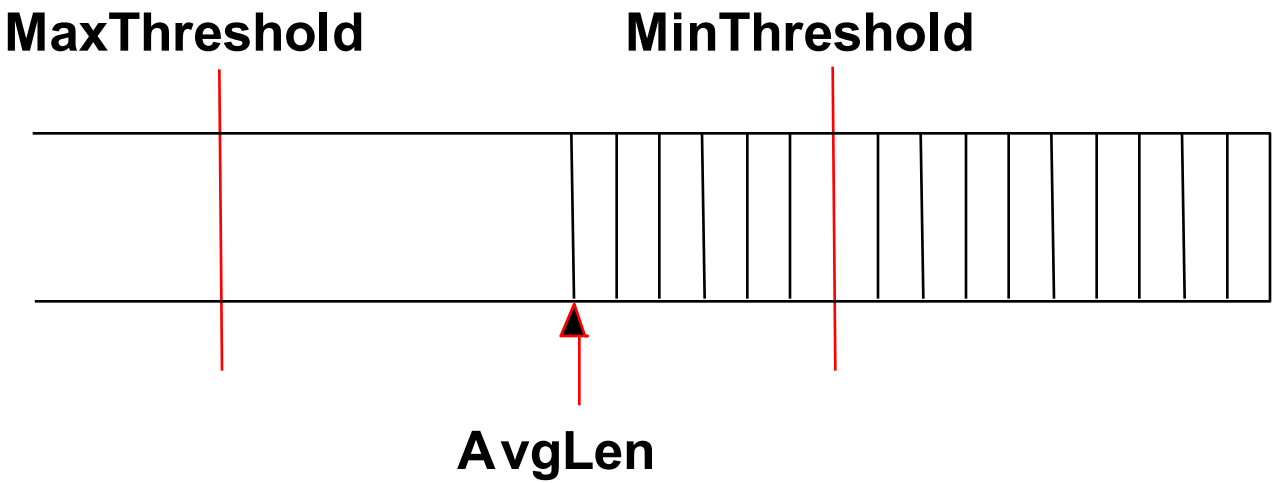
A: RED加权动态平均队长

- 与TCP超时计算中加权值类似：
  - Avglen=(1-Weight)×Avglen+Weight×SampleLen,
  - 其中0＜ Weight ＜1， SampleLen样本测量时长
- 平均队列长度
  - 比瞬间队长更能准确捕获拥塞信息，瞬时队列会突然塞满或腾空，依此来通知主机升降发送速率并不合适
- 加权平均值
  - 是一个低通滤波器（去除高频信号），加权值决定滤波器的时间常数



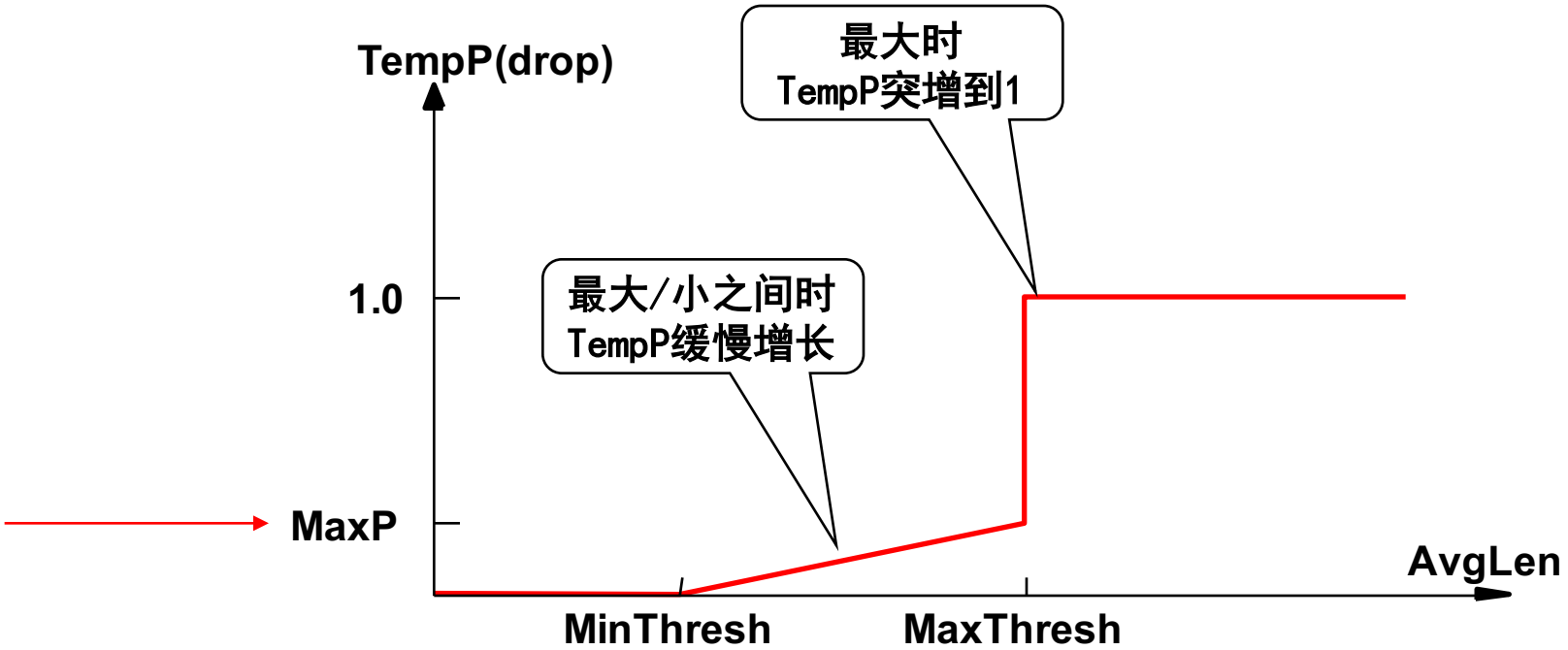
B: FIFO队列长度阈值的决定

- 最小和最大阈值MinThreshold/MaxThreshlod
  - 当包到达时，将当前Avglen与最小和最大2个阈值比较
    - If Avglen ≤ MinThreshold then queue packet
    - if MinThreshold ＜ Avglen ＜ MaxThreshlod then calculate probability p and drop the packet with probability p
    - if MaxThreshlod ＜ Avglen then drop the arriving packet



主动丢包概率函数

- 实际丢包函数P
  - TempP是图y轴表示的变量，是一个小于MaxP的值
  - TempP = MaxP * (Avglen-MinThreshold) /(MaxThreshold- MinThreshold)
  - P = TempP/(1-count×TempP)，count记录从上一次丢包开始到现在有多少刚到的包已加入队列，所以P不仅和Avglen有关，而且还和新进入队列的包的数量count有关。随着count的增加，下一个包被丢弃的可能性也在缓慢增加。
  - 这样可使丢弃随时间分布，更早的通过重复ACK通知发送端，避免慢启动

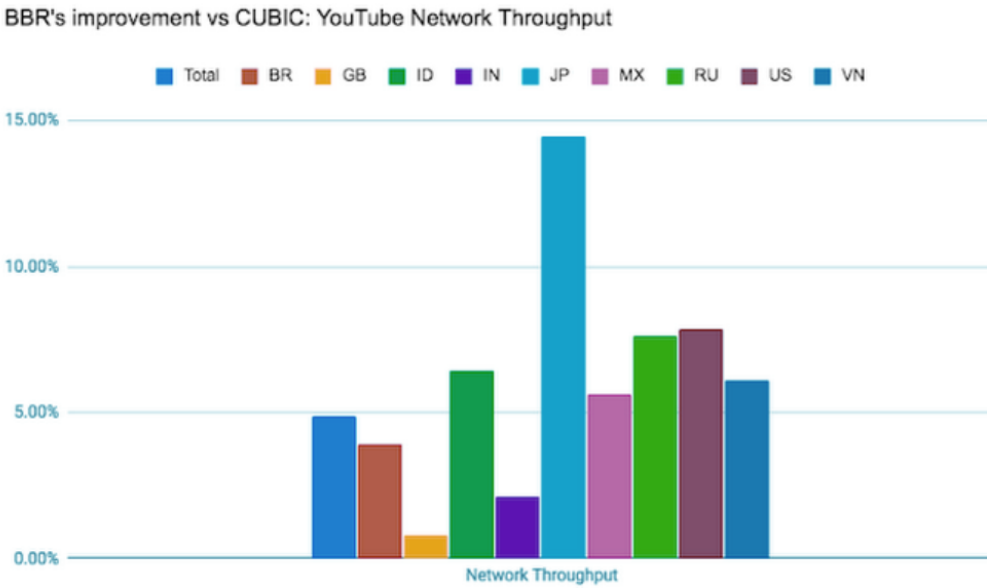


RED丢包概率随时间均匀分布

- 假定初设置
  - MaxP=0.02,count=0,Avglen位于最大最小值的中间值Δ,
  - 则TempP= MaxP/2=0.01即TempP = 0.02 ×（Min+Δ/2-Min）/Δ=0.01， P = 0.01/(1-0),
  - 包有99%概率进队列
- P缓慢增加,
  - 当50个包到达后,P=0.01/(1-50*0.01)=0.02
  - 当count=99时(中间没有丢包),P=0.01/ (1-99*0.01)= 1,保证下一分组一定丢失
- 算法重点
  - 保证丢弃概率大致随时间均匀分布
  - 丢弃某特定流包的概率和该流在R上获得的带宽分额成比例--保证基本平均分配

附加：TCP的BBR 拥塞控制算法（自学）

- Google 2016年公开的BBR算法，BBR: Congestion-Based Congestion Control
- 2700x faster than previous TCPs on a 10Gb, 100ms link with 1% loss
- Youtube应用后在吞吐量上有平均4%提升（对于日本这样的网络环境有14%以上的提升）



BBR 算法

TCP BBR 的目标：

- 在有一定丢包率的网络链路上充分利用带宽。
- 降低网络链路上的 buffer 占用率，从而降低延迟。

BBR认为TCP congestion control是在1980s提出的，当时认为丢包等于拥塞的假设并不是天然正确，而是由于技术局限。当进入Gbps传输时代，丢包和拥塞之间的关系变得很不明显了。

BBR 算法

标准 TCP 的做法有两个问题：

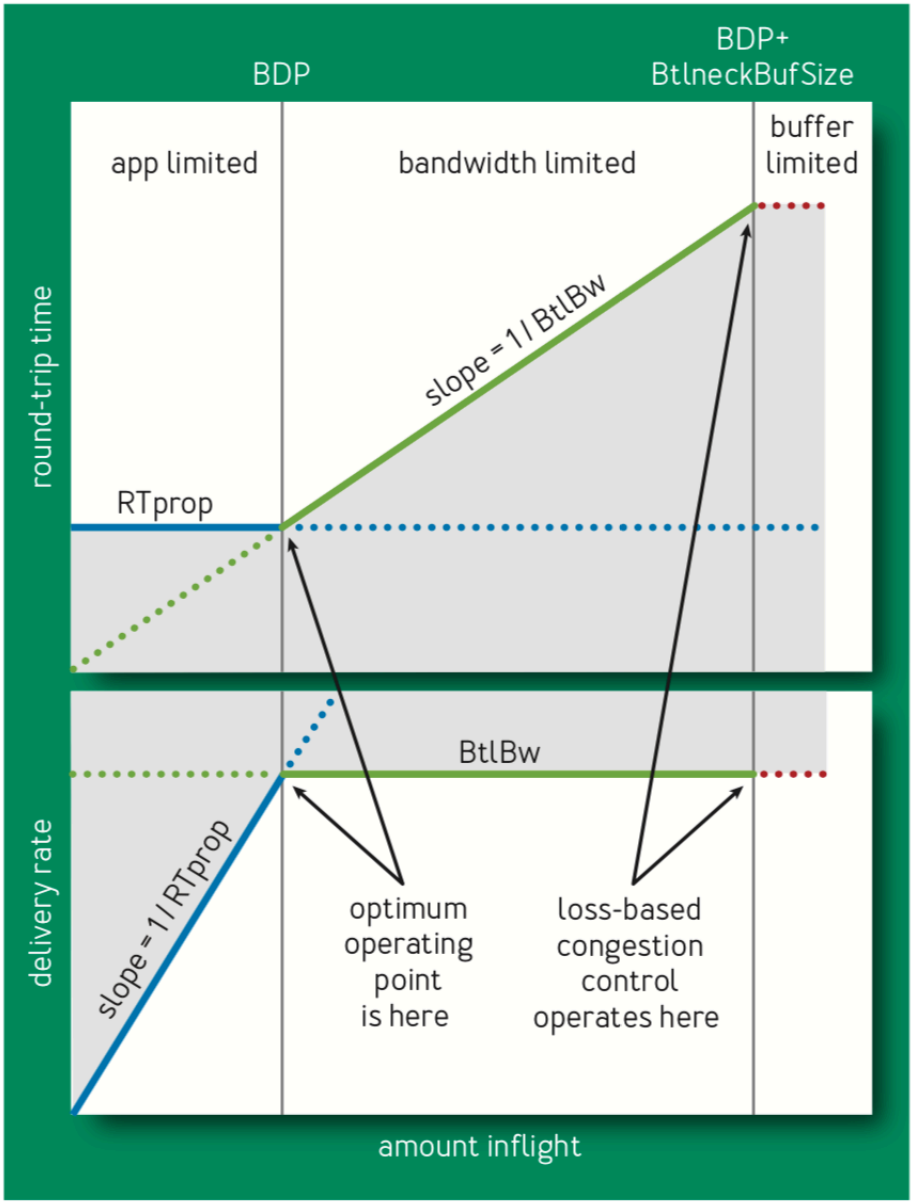
- 标准 TCP 是通过 “灌满水管” 的方式来估算发送窗口的，在缓冲区较大的链路上，TCP会一直发送导致缓冲区处于较满状态，导致发送延迟增加。这个问题被称为 bufferbloat。
- 但是在缓冲区较小的链路上，TCP认为丢包是拥塞，这样就导致了很小的CW窗口，吞吐率较低。

为了改变这种情况，BBR提出要改变loss-based congestion control。而是真实测量一条链路上的拥塞: bottleneck bandwidth and round-trip propagation time, or BBR，目标是达到BDP (= BtlBw × RTprop)

BBR如何优化呢？

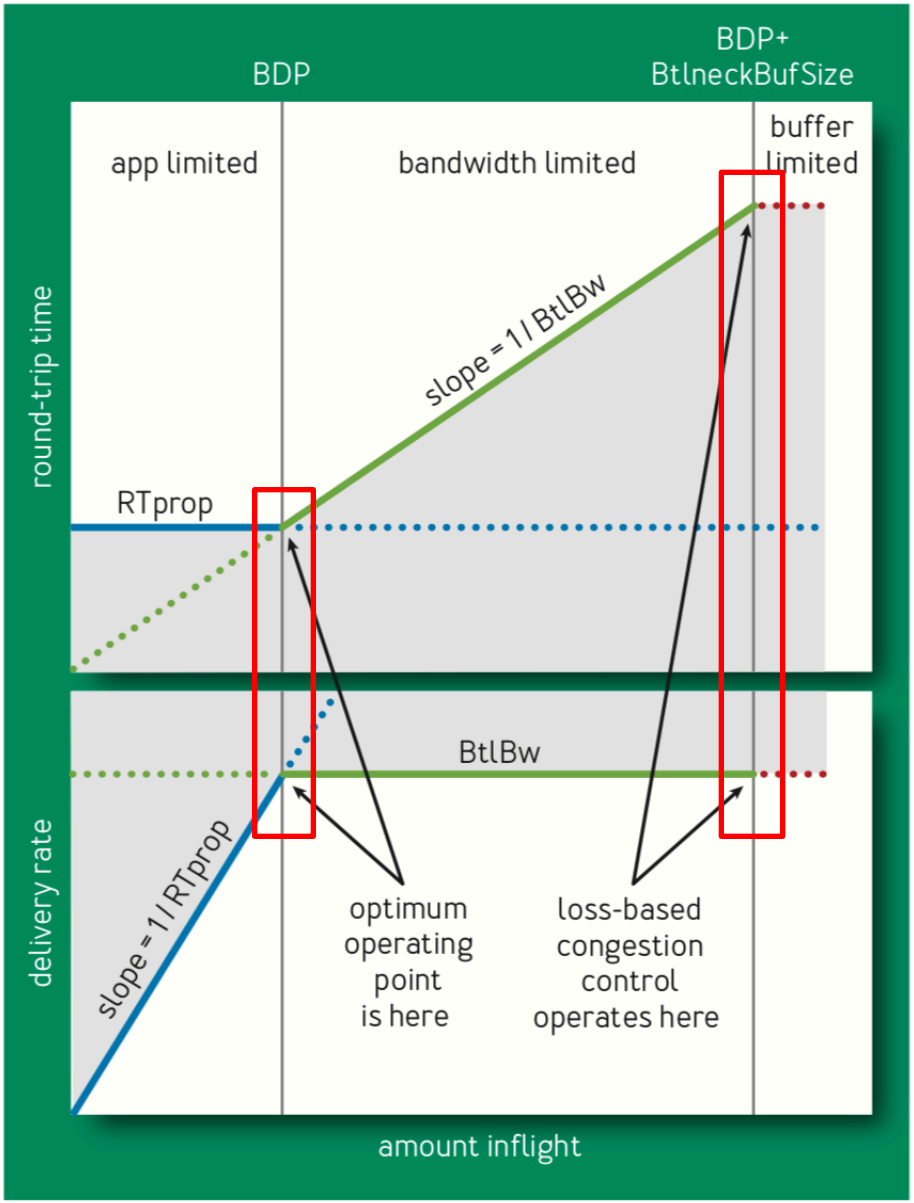
网络中报文数和RTT以及发送速率关系图，图中有两个子图，横坐标都为在网络中的数据，纵坐标为发送速率和RTT：

- Rtprop 就是RTT传播时间（RTT是平均值）
- BtlBW瓶颈带宽
- 蓝色的虚线线表示RTT限制
- 绿色的虚线线表示瓶颈带宽限制
- 红色的虚线线表示瓶颈缓冲区
- 下面子图中的实线表示发送速率变化
- 上面子图中的实线表示RTT变化



BBR如何优化呢？

- 当网络中数据包不多，还没有填满瓶颈链路的管道时，随着投递率的增加，往返时延不发生变化。
- 当数据包刚好填满管道，达到网络工作的最优点（满足最大带宽BtlBw和最小时延RTprop），则在最优点网络中的数据包数量=BDP。
- 继续增加网络中的数据包，超出BDP的数据包会占用buffer,达到瓶颈带宽的网络的投递率不再发射变化，RTT会增加。继续增加数据包，buffer会被填满从而发生丢包。故在BDP线的右侧，网络拥塞持续作用。



BBR 算法

1. 不考虑丢包
2. 估计最优工作点 (max BW, min RTT)
3. 利用最优工作点作为发送率，不考虑Congestion Window，更多的是考虑发送缓冲区限额（习惯也称为CW）
4. 一开始有慢启动和指数回退，但是一旦进入稳定发送状态就不再加法增加/指数减少了
5. max BW和min RTT不能被同时测得。要测量最大带宽，就要把瓶颈链路填满，此时buffer中有一定量的数据包，延迟较高。要测量最低延迟，就要保证buffer为 空，网络中数据包越少越好，但此时带宽较低。BBR的解决办法是：交替测量带宽和延迟，用一段时间内的带宽极大值和延迟极小值作为估计值

实验3 TCP拥塞控制

- 通过实验，加深理解TCP拥塞控制机制，包括FIFO和RED等排队、丢包策略
- 报告格式跟实验2一样
- 2019年12月25日之前交