2.7 TCP拥塞控制(源抑制)

- 源算法中使用最广泛的是TCP协议中的拥塞控制算法. TCP是目前在Internet中使用最广泛的传输协议.
- 根据MCI的统计, 总字节数95%和总报文数的90%使用TCP传输.
- 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）
Reno-TCP	加上快速恢复. （当前最为广泛采用的TCP实现版本）
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.

- TCP基本策略：
  - 端端拥塞控制
  - 无任何预约把包发到网络，后观察事件反应
- TCP假设网络中R仅用FIFO排队；但也用FQ规则
- 拥塞包在R中丢弃后，由重传解决
- TCP又叫自定时（self clocking）
- 以下描述3个TCP拥塞控制机制

2.7.1 加法式增加/乘法式减少

- CongestionWindow 是源端TCP为每个连接维护的一个状态变量：
  - 以限制在给定时间内容许传输的数据量
  - 是流控AdvertisedWindow的一个副本
- 容许的最大未确认字节数就是现在的最小CongestionWindow和AdvertisedWindow
  - MaxWindow = MIN(CongestionWindow, AdvertisedWindow)
  - EffectiveWindow = MaxWindow - (LastByte - LastByteAcked)

TCP怎样学到CW

- AW靠连接的接收方来发送，没有谁发送CW到发送方
- 答案：TCP源根据它从现在网络觉察到的拥塞程度来设置CW
  - 当拥塞程度上升时，就减少CW
  - 当拥塞程度下降时，就增加CW
  - 上述二者合在一起就叫：加法式增加/乘法式减少

怎样决定是否拥塞？

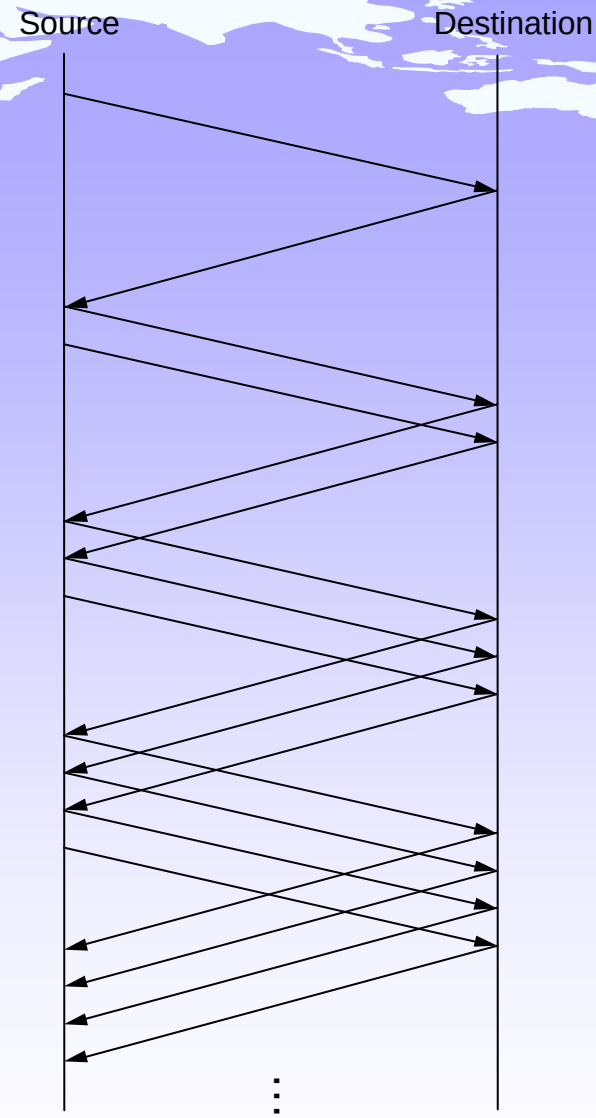
- 观察主要原因：包未交付，并导致超时结果，就是由于拥塞丢包，由于传输错误丢包是很少的
- TCP把超时解释为拥塞的信号并降低传输速率
- 特别，每次超时发生，**TCP源把原先的CW值的一半作为CW现值**，这就是乘法减少机制的一部分

折半减少CW

- 虽然CW定义为字节单位，但也容易以包为单位考虑。例如：
- CW现在是16个包，若检测到丢包，则设置CW ← 8 = 16/2，再丢一次包将为8/2 = 4，4/2 = 2，最后到2/2 = 1；TCP不再允许比1个包更小的CW，或最大段尺寸（MSS）

加法式增加CW

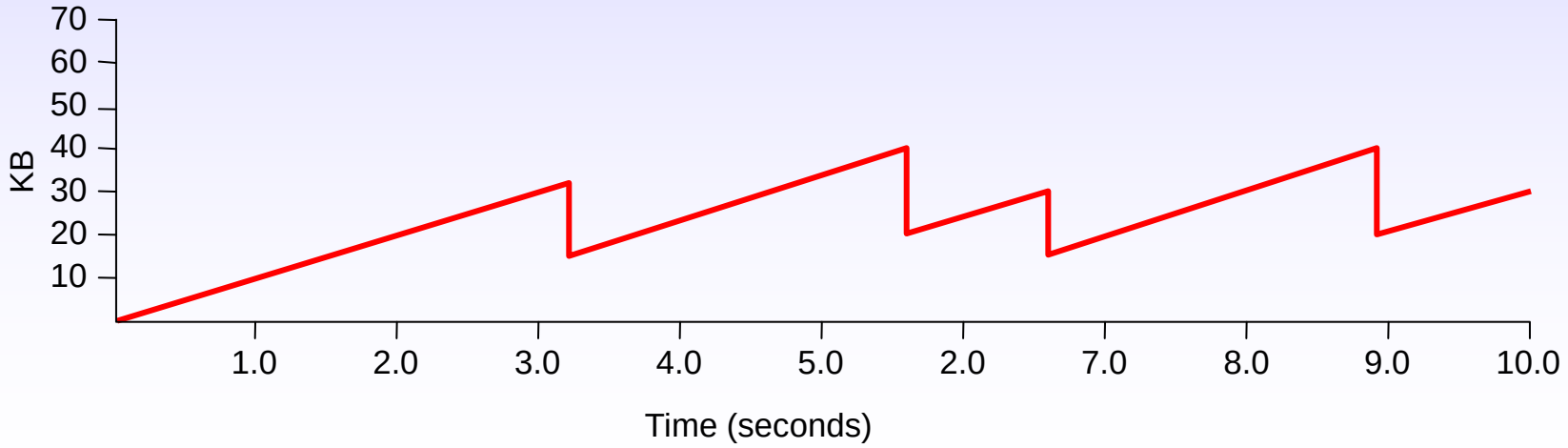
- 当TCP源每次成功发送CW个包后即最后一个RTT中每个包都收到了其ACK，则 CW ← CW+1，这种线性增加如图2.8所示：



在每个RTT后加1个包

实际是锯齿模式加

- 实际并不是等待全部CW个包的ACK都到后才cw←cw+1，而是在每一个ACK到达后给CW增加一点，其增量为
  - Increment = MSS×(MSS/CW)
  - CW += Increment

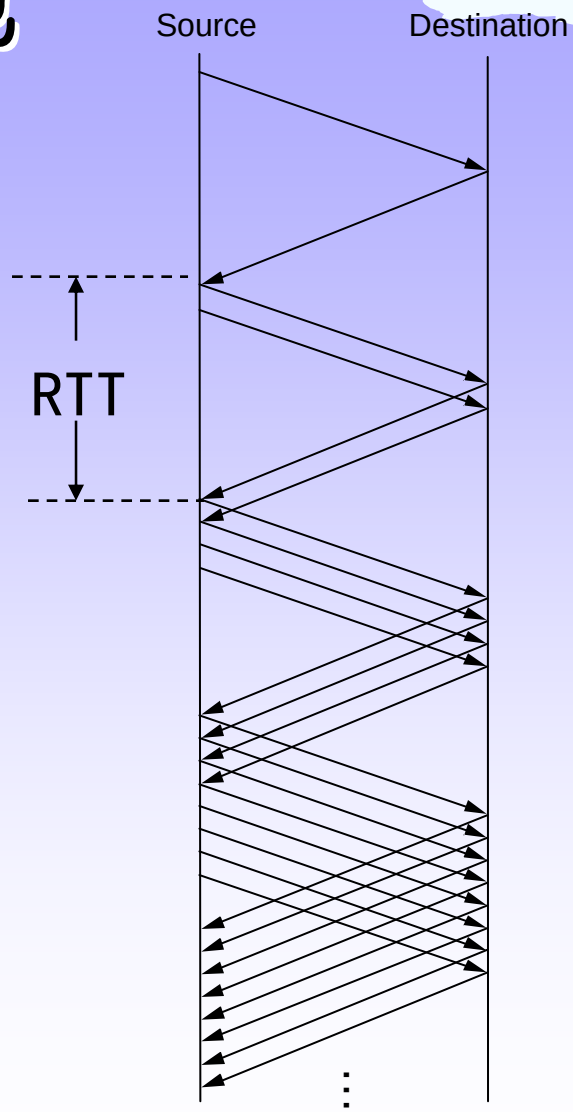


2.7.2 慢启动

- 上述的线性加对运行在接近网络可用容量的源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提出第2个机制—缓慢开始（讽刺）：指数式增加CW，而不是线性。
  - 开始，源把CW置为1（一个包）；
  - 当该包的ACK到达后，TCP把CW加1并发2个包；
  - 当收到2个ACK后，CW+2，每个ACK一个，但发4个包，故每个RTT内包数翻一倍

缓慢开始传输中的包

- 指数增长反叫缓慢—这首先是个迷
- 但从历史看，不要同线性比较，而同TCP原来的速度（未考虑拥塞控制时更快）
- 该方法未出现前，源按Advertised Window可立刻发送所有数据



缓慢的2个情况

- 每个连接的开始，源并不知道在给定时间内能传输多少个包（TCP能在9600bps-2.4Gbps链路上运行，故源无法知道现行网络的容量）
- 等待超时发生时，连接趋向终止，使用缓慢开始更敏感。源缓慢重新开始数据流，而不是把全部窗口数据立刻发送到到网络。**类似TCP滑动窗口**：若丢包，源发送了通告窗口所容许那些数据，并阻塞，等还未的ACK，最终超时发生，但直到此时也没有包传输，即源将收不到ACK，故也不能传输新的包 。源将接收一个积累的ACK（它重新打开的整个通告窗口）

TCP增加CW的代码

```
{
 u_int cw = state->CongestionWindow;
 u_int incr = state->Maxseg;

 if(cw > state->CongestionThreshold)
     incr = incr * incr/cw;
   CongestionWindow=MIN(cw+incr,TCP_MAXWIN);
}
```

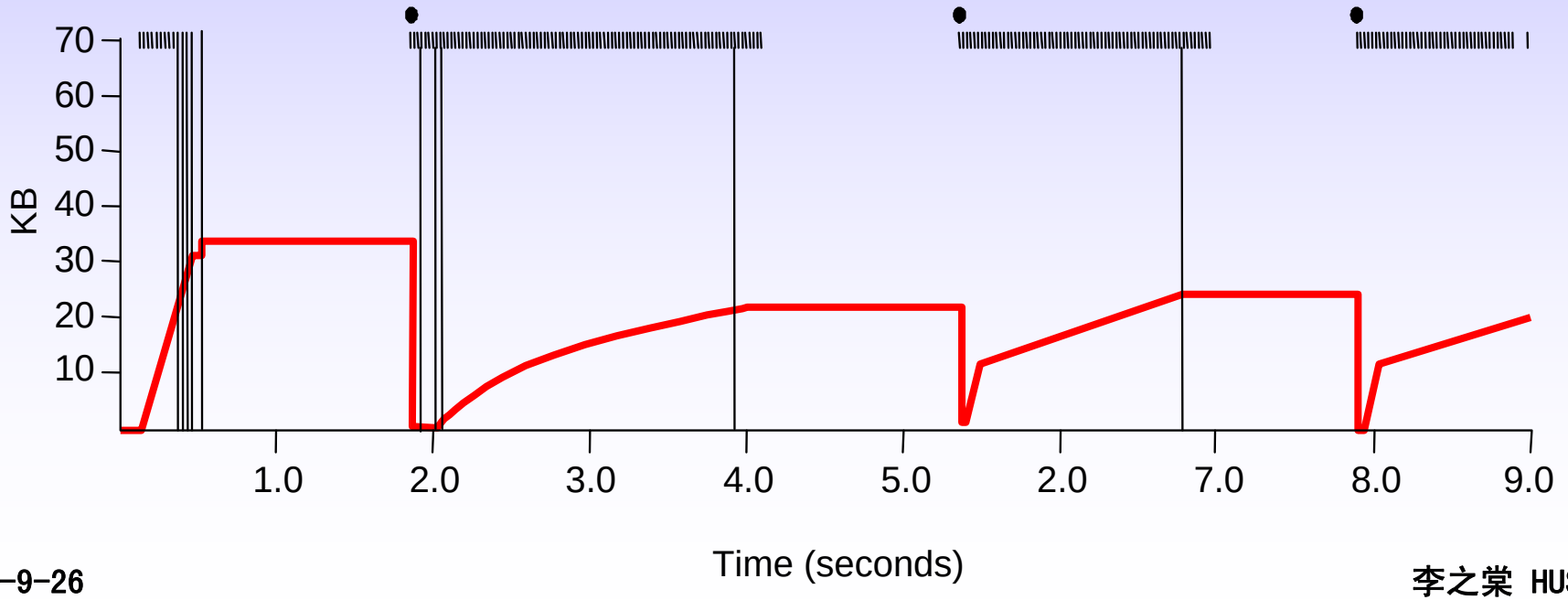
- State:一个TCP连接的状态； TCP_MAXWIN：容许的CW上界； Maxseg：最大段长

跟踪实际的TCP连接

- 几个注意事项：
- 连接开始，CW快速增加，这慢开始阶段的初始化持续到进入连接约0.4秒内丢几个包，此时CW变平约34KB，原因是没有ACK到达，并有丢包，此时也不会有新包发送，大约2秒后超时最终发生，CW/2，即从34KB减到约17KB，拥塞阈值（CT）设置成该值。慢开始将重设一个包并从这儿开始新的斜坡

TCP拥塞控制动作

- 红线 = CW的值；顶上子弹头 = 超时；顶上分离粗点标记 = 每个包发送的时间；垂直线条 = 最终重传而第一次被发送的包



4--5.5--8 秒阶段

- 2秒时丢失2个包，故在2--4秒间CW应是线性增加。4秒后CW平缓。在5.5秒处又一次丢包，此时：
  - 1、超时发生，导致CW=CW/2，从22KB-->11KB，CT<--11KB
  - 2、CW从一个包开始启动，发送方进入慢开始阶段
  - 3、慢开始引起CW指数增长直到达到CT
  - 4、CW开始线性增长
- 在8秒等其它超时点也重复这相同的模式

慢开始阶段为什么会丢包？

- TCP此时要学到网络上多大带宽是可用的，这是一非常困难任务
  - 若源此时不积极，如其仅线性增加CW，故它需要较长时间来发现可用的带宽，这对本次连接的吞吐率有很大影响
  - 若源此时是积极，则TCP的CW是指数增长，则源存在CW/2包要被网络丢包的风险

替代慢开始的其它策略

- 有些网络设计者提出：Packet pair(包对)策略---TCP源试图通过更聪明的发出一组包有多少通过了来估算可用带宽。基本思想是：
  - 发送2个中间没有间隔的包，TCP源观察：这2个包的ACK离开有多远
  - 2个ACK间的这一间隙作为网络拥塞的测量，故增加多少CW是可能得出得

2.7.3 快速重发和快速恢复

- 把拥塞控制加到TCP后马上发现:超时得粗略实现使在等待超时时间期间连接死
- 快速重发是启发式,丢包重发的触发有时比常规超时机制要快
- 快速重发机制并不取代常规超时,只是提高

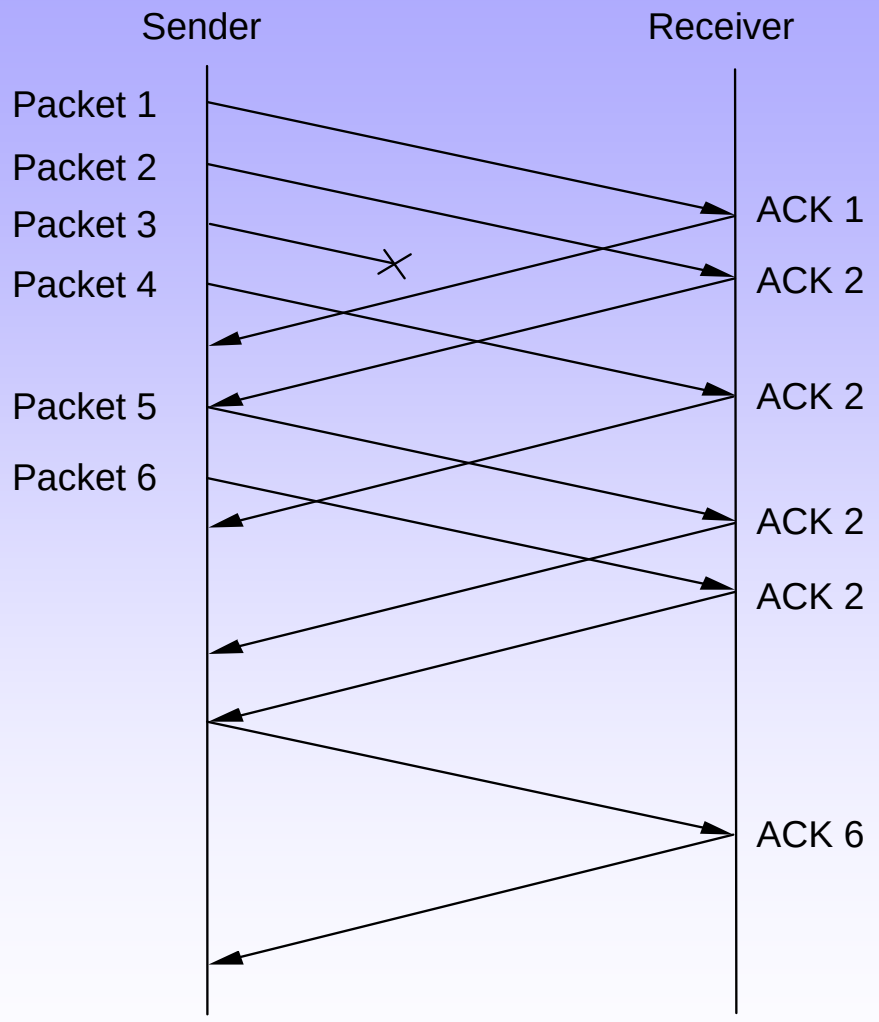
快速重发机制基本思想

- 每次一个数据包到达接收方,接收方响应一个ACK,即使这个序列号已经确认.
- 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,这说明比该包更早的包可能被丢失
- 实践中,TCP等到3个失序包后就重发丢失的包

下图说明重复ACK导致快速重发

- 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
- 当包5,包6到达时,目方仍重发包2的ACK
- 当发方收到包2的第3个ACK后,它重发包3
- 当重发的包3到达目方后,收方返回一个累计ACK,含有包6
- 下图中没有重发表示

基于重复ACK的快发重发机制

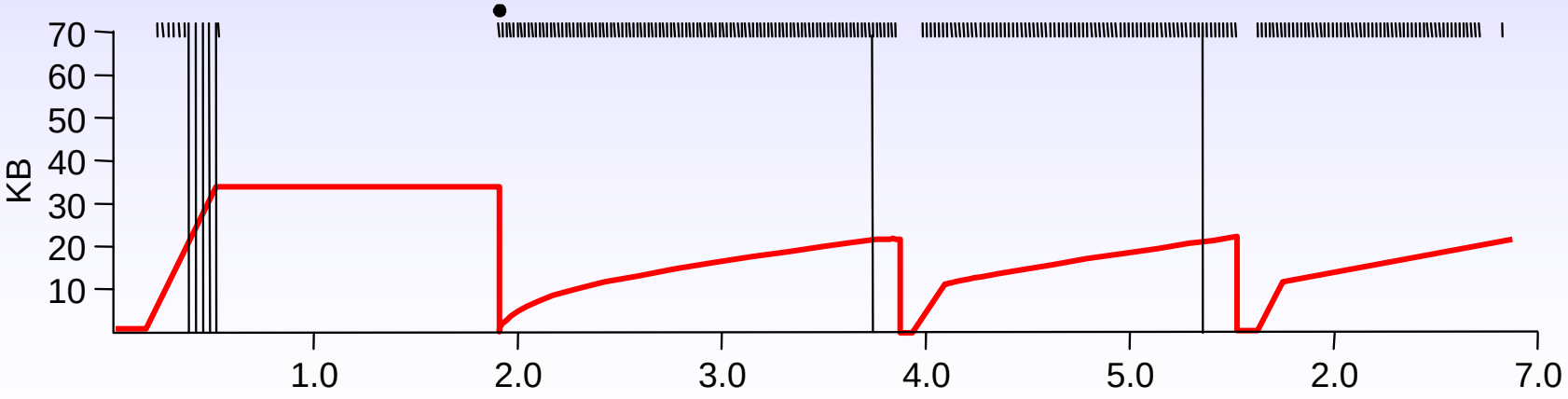


改进

- 当快速重发机制提示拥塞时,与其减少CW回到1并又进行慢开始,还不如用仍在传送中的ACK来对发送包计时---称为快速恢复
- 该机制能有效消除:快速重发检测到一个丢失包和加式增加开始之间的慢开始阶段:例,快速恢复避免了图2.11中3.8--4秒之间的一段慢开始,并销减CW一半(从22KB到11KB)并重新开始加式增加

带快速重发的踪迹

- 红线 = CW的值；顶上子弹头 = 超时；顶上分离粗点标记 = 每个包发送的时间；垂直线条 = 最终重传而第一次被发送的包
- 换句话说:慢开始仅仅在连接的开始和粗粒度超时发生时,而在所有其它时间,CW是纯线性增加/乘式减少模式



2.8 拥塞避免机制

- 应该懂得：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。换句话说，**TCP需要创造丢失来发现该连接的可用带宽**。
- **拥塞避免策略**（但未被广泛采用）：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**
- 3个不同的拥塞避免机制：前2个相似：把少量功能件加入到路由器中，帮助端节点预测拥塞；第三个是纯粹由端节点来避免拥塞

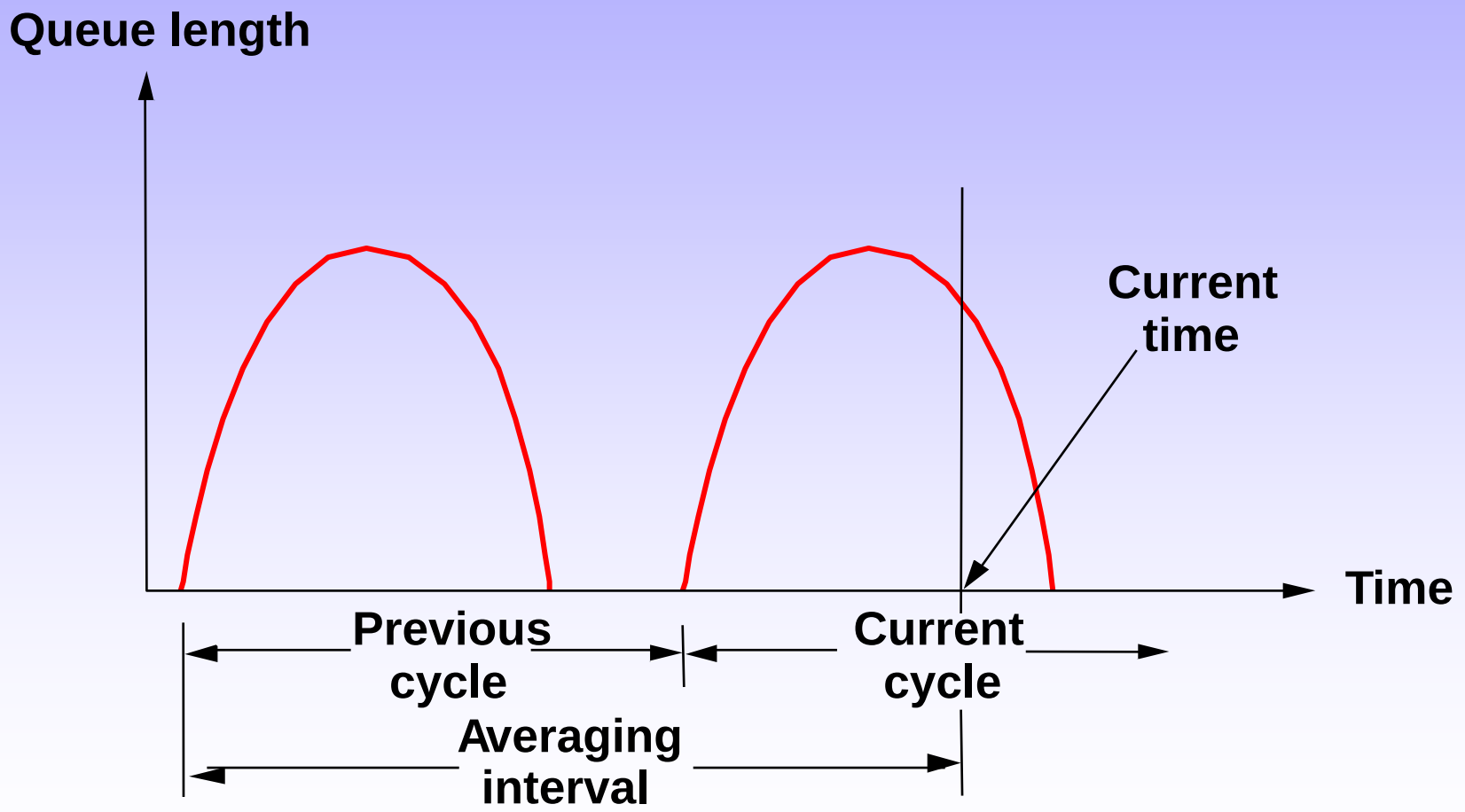
2.8.1 DECbit

- **第一种机制**在Digital Network Architecture上开发并使用，故也能用在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，因由DEC公司开发，故称 DECbit
  - 目端把这拥塞位拷贝进ACK并返送回源端
  - 源端调整自己的发送速率以避免拥塞

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
- 测量平均队长的时间间隔是：最近的忙期+空闲期+当前忙期（R传输时为忙，不传输时为闲），图2.14显示路由器中时间与队长的函数
- 路由器计算曲线下的面积并除以时间间隔得到平均队列长度
- 用队长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是能力（**吞吐率/延迟**）函数的优化

路由器中平均队长的计算



Chapter 6, Figure 14

另一半机制--主机的工作

- 源将记录有多少包导致其在路由器中记下拥塞位
- 实际上源维护一个拥塞窗口，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

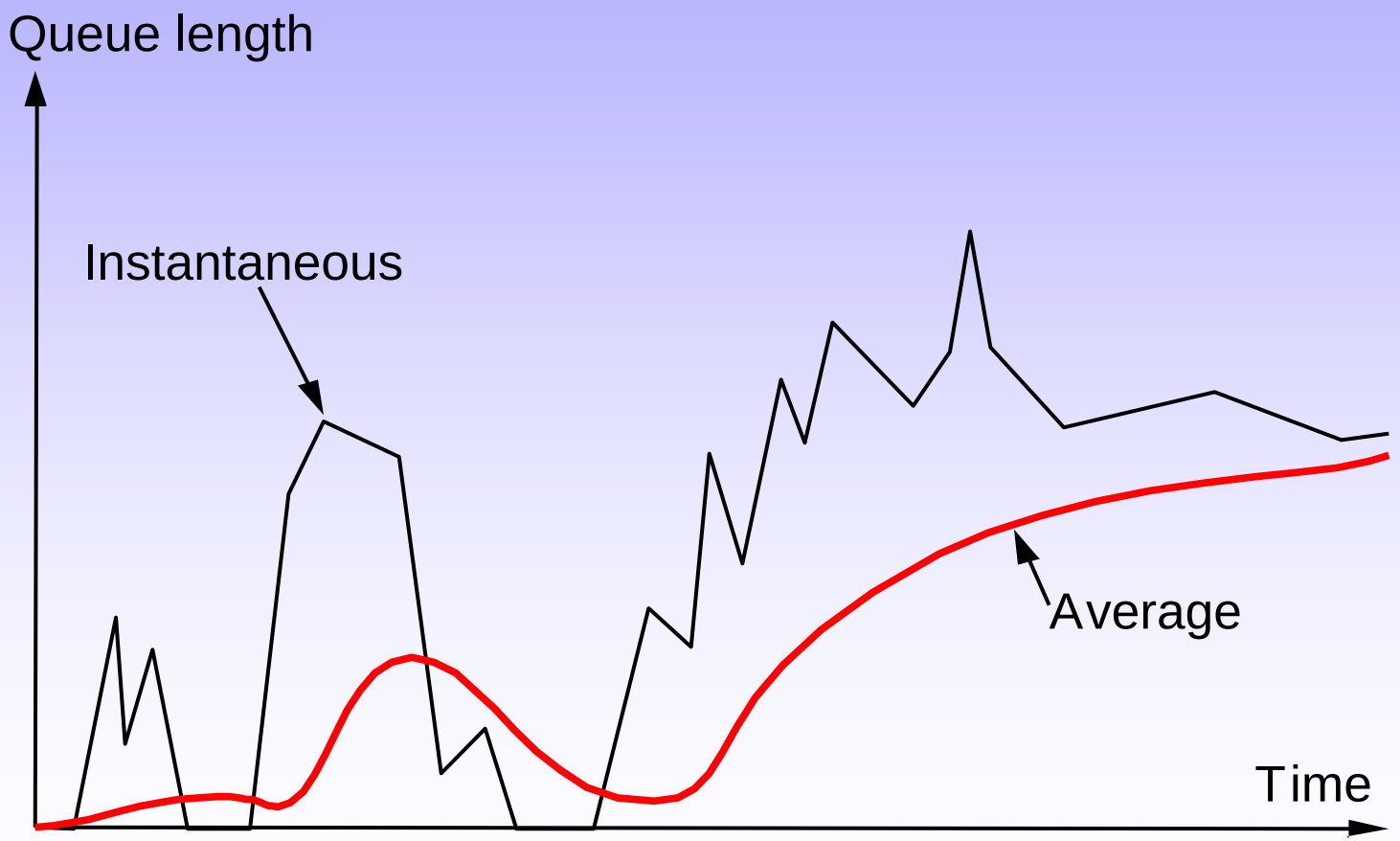
2.8.2 随机早检测

- **第二种机制**-RED：与DECbit相似，都是在每个R上编程监视自己队长，检测到拥塞就通知源调整窗口
- Early:R在其缓冲被完全填满前就丢少量包，使源方慢发送速率
- RED与DECbit的不同
  - RED不明确发拥塞通知到源，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--early random drop

A:加权动态平均队长

- 这与TCP超时计算中的加权值类似：
- Avglen=(1-Weight)×Avglen+Weight×Samplelen,其中0＜ Weight ＜1， Samplelen样本测量时对长
- 平均对长比瞬间对长更能准确捕获拥塞信息，瞬时队列会突然塞满或腾空，依此来来通知主机升降发送速率并不合适
- 可认为加权平均值是一个低通滤波器，加权值决定滤波器的时间常数
- 图2.15右半部分表示塞满或腾空

加权动态平均队长

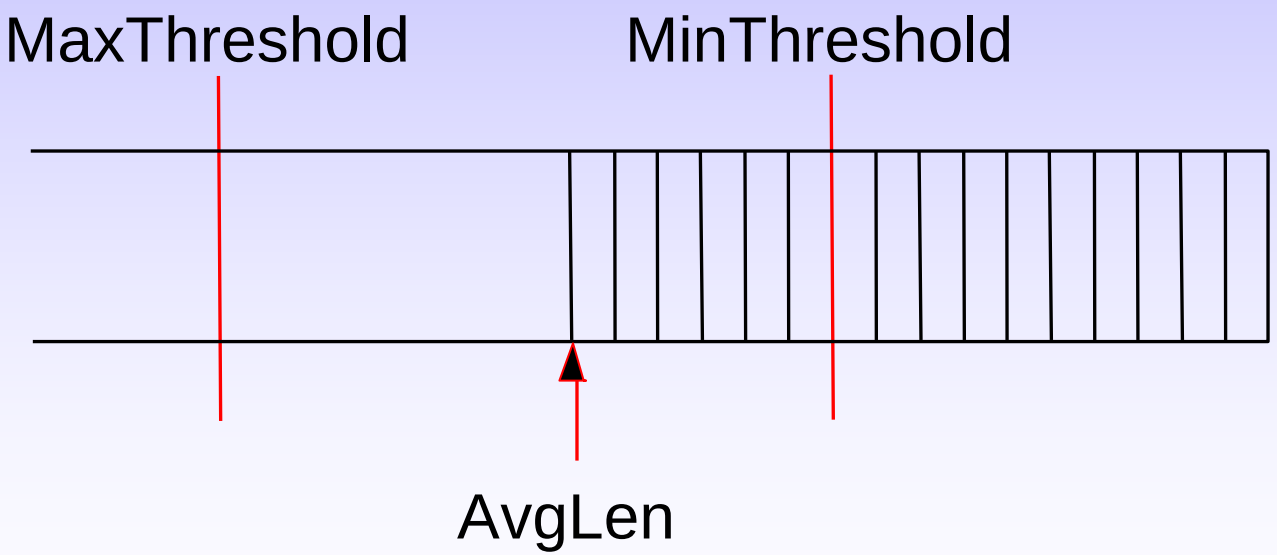


Chapter 6, Figure 15

B：队长阈值的决定

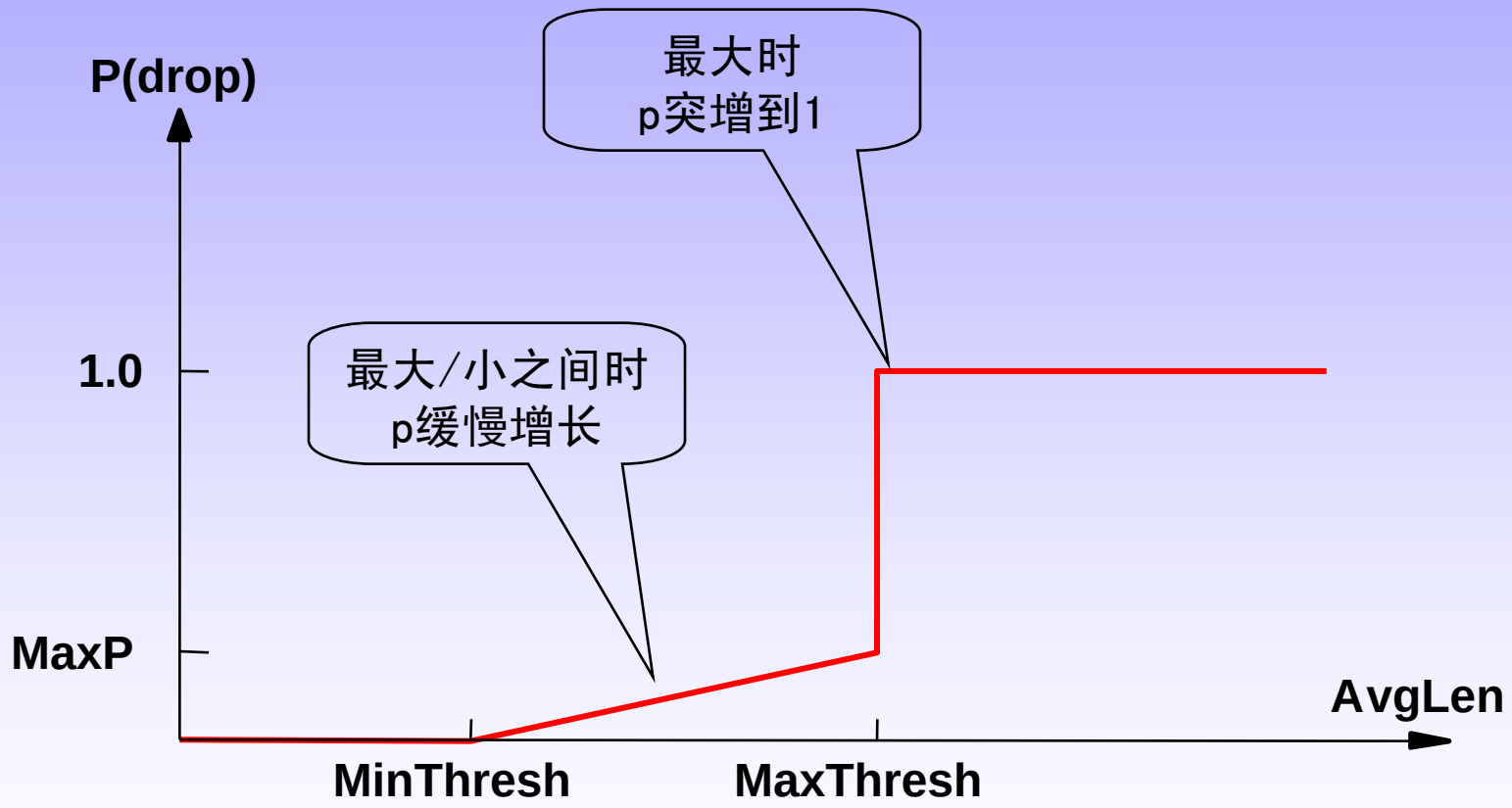
- MinThreshold/MaxThreshlod最小和最大阈值：当包到达时，RED将当前Avglen通最小和最大2个阈值比较
  - If Avglen ≤ MinThreshold then queue packet
  - if MinThreshold < Avglen < MaxThreshlod then calculate probability p and drop the packet with probability p
  - if MaxThreshlod < Avglen then drop the arriving packet

FIFO 排队中的RED阈值



Chapter 6, Figure 16

RED的主动丢包概率函数



Chapter 6, Figure 17

RED的主动丢包概率函数

- 实际丢包函数会更复杂些，事实上p是Avglen和上个包被丢后距当前时间的函数，计算公式如下：
  - TempP = MaxP×（Avglen-MinThreshold）/(MaxThreshold- MinThreshold)
  - P = TempP/(1-count×TempP)
  - TempP是图2.17中y轴表示的变量
  - count记录有多少刚到的包已加入队列
- 这可使丢弃随时间分布，防止单个连接的包成群丢失，从而导致慢启动

RED丢包概率随时间均匀分布

- 假定初设置MaxP=0.02,count=0,Avglen位于最大最小值中间,则TempP(P的初值)= 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(**中间可能出现丢包**),P=0.01/ (1-99*0.01)=1,保证下一分组一定丢失
- 该算法重点保障丢弃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

RED阈值等的考虑

- MinThreshold设置值应足够大得使链路利用率在可接受的高度
- 两阈值之差应大于一个RTT中算出的平均对长的典型增值,当今因特网Max=2Min较合适,且高负载期间,对长在2个阈值之间
- 应有多于MaxThreshold的足够空间来容纳突发性包
- 从R丢弃一包开始到该R减少其窗口而使相应连接减轻压力止,至少要用该连接的1个RTT
- R对拥塞作出反应的时间比通过R的连接的往返时间应少多少?因特网平均RTT=100MS
- 故应选Weight使能滤除小于100ms时间内队长变化

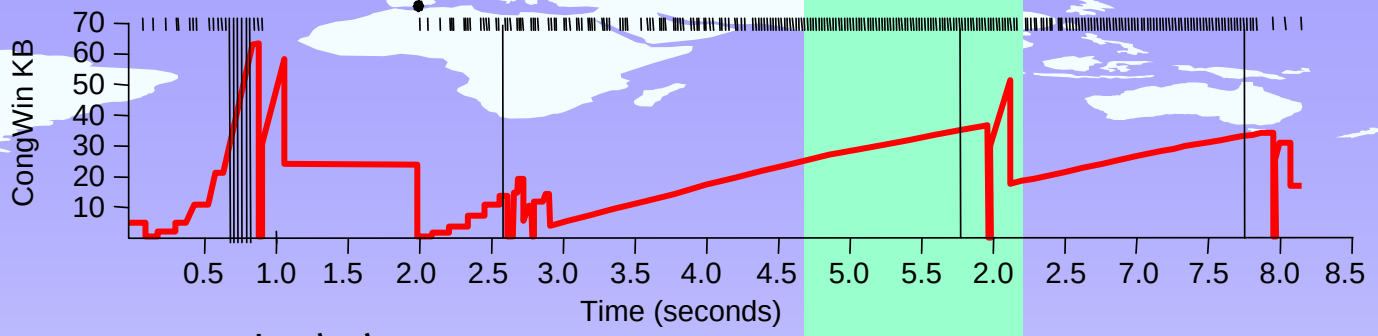
2.8.3 基于源的拥塞避免

- 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，即和刚建立连接时状态一样，则窗口减1．通过计算未发完字节数/RTT得吞吐量

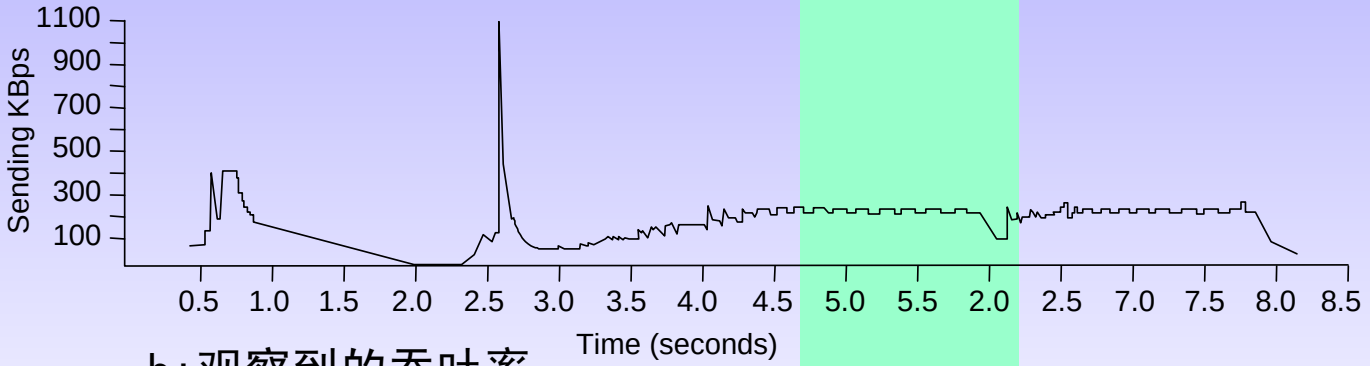
算法4:TCP Vegas

- 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**
- 3图同步,图6-18.a 显示拥塞窗口的变化,b显示在源上测得的平均发送率,c图显示在瓶颈R上测得的平均对长
- 在第4.5--2.0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),
  - 然它却保持平缓(b),这是因吞吐率不能超过可获得的带宽,若超过该点
  - 窗口的增加只会导致包在瓶颈R上占用缓冲空间(c)

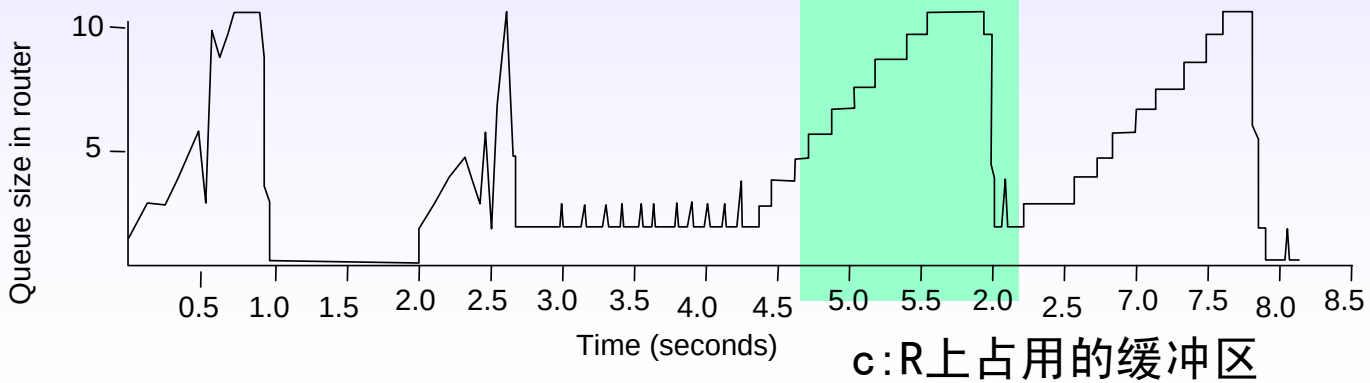
- 拥塞窗口与观察到的吞吐率(3图同步)
- 有色线=拥塞窗口
- 上方黑点=超时
- 散列符号=每个包被发送的时间
- 竖条=最宗将被传送的包的第一次传送时刻



a:拥塞窗口



b:观察到的吞吐率



c:R上占用的缓冲区

Chapter 6, Figure 18

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，也是R溢出前该连接发送的第一包的RTT，希望吞吐率为
  - ExpectedRate=CongestionWindow/BaseRTT
  - 讨论方便假设CongestionWindow等于传送中字节数
- Step2：计算当前发送率ActualRate,即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:TCP Vegas比较ActualRate和ExpectedRate,并相应调整窗口,我们令:
  - 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的额外数据
  - 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β使发送率应下降;而当二者太接近时,该连接可能不能充分利用带宽, α使窗口增加

TCP Vegas拥塞避免机制的过程

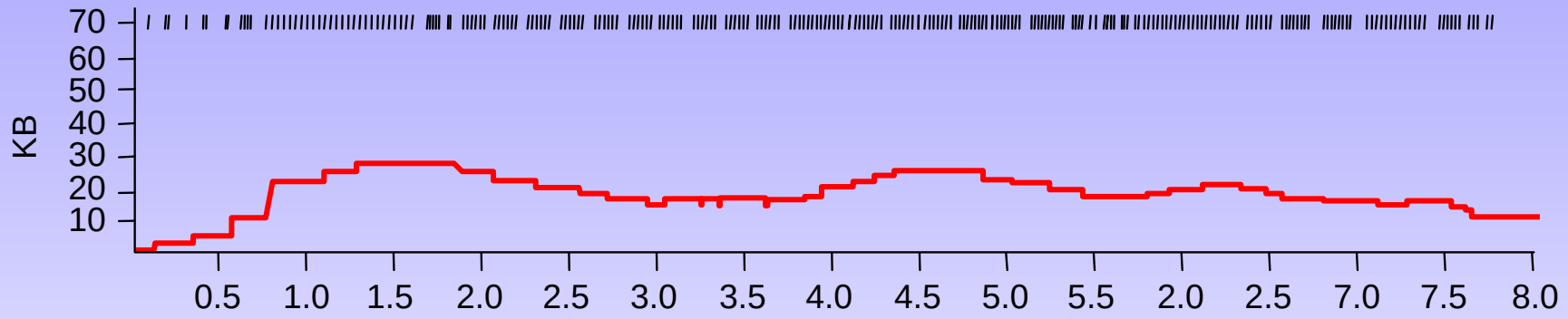


图2. A:拥塞窗口

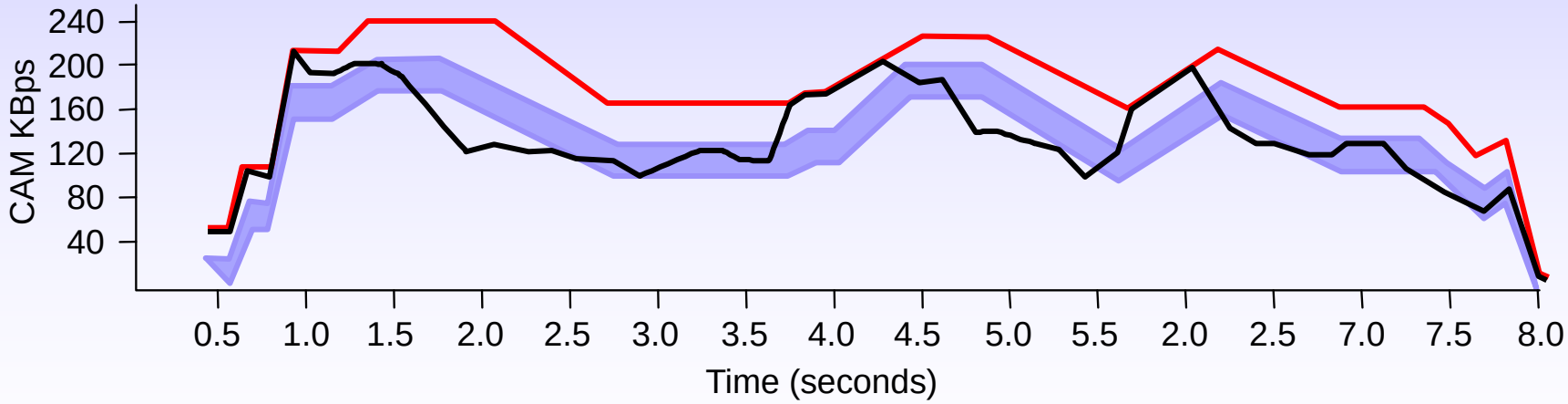


图2. B:期望(红色)和实际(黑色)吞吐量,阴影区是α,β 阈值中间的区域

阈值α β的单位与大小

- 阈值的单位是KB, 如某连接, 其BaseRTT = 100 Ms, 一个包长= 1KB, 若α = 30KBps, β = 60KBps
- 故可为该连接在网络中说明α至少占用3个额外buffers, β占用6个额外buffers
- 实际使用中α设置为1个缓冲区, β设置为3个缓冲区时性能较好

Thank you!