主要内容

- **网络资源管理（续）**
  - **数据中心负载均衡技术: ECMP, MPTCP（补充）**
  - **传输层的拥塞控制**
    - **TCP拥塞控制**
    - **TCP协议（复习）**

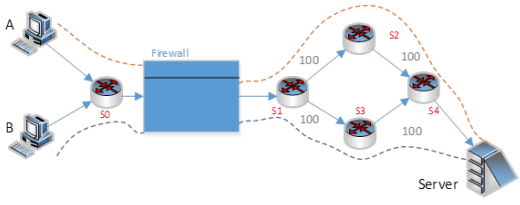
数据中心的负载均衡技术

数据中心的负载均衡需求

- 数据中心存在大量的路径资源
- 数据中心最常使用的负载均衡算法：ECMP
  - 通过根据数据流的五元组哈希，将这些数据均匀随机的分散到权重相等的路径上。这种随机选路负载均衡可能产生哈希碰撞。
- 使用MPTCP进行数据中心的负载均衡
  - 在利用多路径的同时，还可以对流量进行拥塞控制，动态的将数据更多的发送到负载低的链路上。有效的提高负载均衡性能。

ECMP

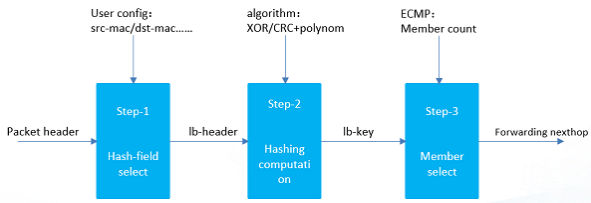
- **ECMP, Equal-Cost Multi-path，等价多路径**
  - 存在多条到达同一个目的地址的相同开销的路径。
  - 当设备支持等价路由时，发往该目的 IP 或者目的网段的三层转发流量就可以通过不同路径分担，实现网络的负载均衡
  - 当某些路径出现故障时，由其它路径代替完成转发处理，实现路由冗余备份功能。



ECMP的路由选择策略

- 哈希
  - 根据源IP地址的哈希值为不同流选择转发路径
- 轮询
  - 各个流在多条路径之间轮询传输
- 基于路径权重
  - 根据路径的权重分配流，权重大的路径分配的流数量更多

ECMP流程



http://www.ruijie.com.cn/fa/xw-hlw/82104/

HASH因子的选择

- **首先数据报文转发查询路由表，确认存在多个等价路由，再根据当前用户配置的流量均衡算法，提取参与 HASH 计算的关键字段，即HASH因子。**

流量均衡模式	HASH 因子
SRC-MAC	IP address source（SIP）
DST-MAC	
SRC-DST-MAC	
SRC-IP	
DST-IP	IP address source and destination（SIP+DIP）
SRC-DST-IP	
SRC-DST-IP-L4PORT	IP address source and destination，L4 port source and destination（SIP+DIP+SP+DP）
Enhanced	增强模式，根据load-balance profile 提取报文字段，可以定义配置已有的hash因子，也可自定义hash扰动因子

HASH计算

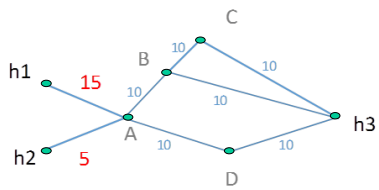
- 根据 HASH 算法进行计算，得出相应的 HASH lb-key(load-balance key)。 ECMP 流量均衡支持的 HASH 算法包括异或（XOR）、CRC、 CRC+扰码等。
- 数据报文经过路由查表后找到对应ECMP 基值（base-ptr），根据 HASH 因子通过 HASH 算法计算获得 HASH lb-key 后，进行 ECMP 下一跳链路数（Member-count ）求余计算，再与ECMP基值进行加法运算得出转发下一跳index，即确定了下一跳转发路由。
- 计算公式：Next-hop =（lb-key % Member-count）+ base-ptr

存在问题

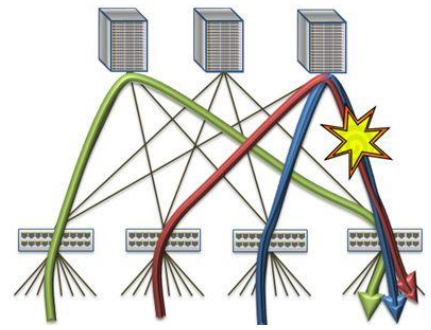
- 可能增加链路拥塞
  - ECMP没有拥塞感知的机制，只是将流分散到不同的路径上转发。对于已经产生拥塞的路径来说，很可能加剧路径的拥塞。而使用哈希的方法，产生哈希碰撞也会增加链路的拥塞可能。
- 非对称网络使用效果不好
  - 例如，A与h3之间的通信，ECMP只是均匀的将流通过B,D两条路径分别转发，但实际上，在B处可以承担更多的流量。因为B后面还有两条路径可以到达h3。
- 基于流的负载均衡效果不好
  - ECMP对于流大小差异较大的情况效果不好

非对称网络使用ECMP

- 例如：A与h3之间的通信，B有两条路径到达h3，D只有1条路径到达h3



拥塞问题



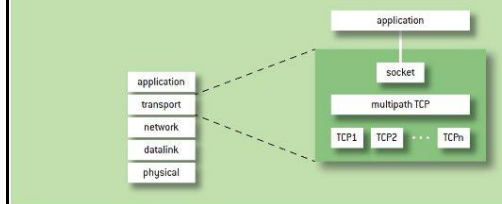
MPTCP

- MPTCP（Multipath TCP），RFC 6824
  (https://tools.ietf.org/html/rfc6824)
- MPTCP允许在一条TCP链路中建立多个子通道
- 当一条通道按照三次握手的方式建立起来后，可以按照三次握手的方式建立其他的子通道，这些通道以三次握手建立连接和四次握手解除连接。这些通道都会绑定于MPTCP session
- 发送端的数据可以选择其中一条通道进行传输
- 设计原则：
  - 应用程序的兼容性，应用程序只要可以运行在TCP环境下，就可以在没有任何修改的情况下，运行于MPTCP环境。
  - 网络的兼容性，MPTCP兼容其他协议。

MPTCP协议栈

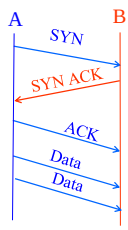
- 相同形式socket API
  - 五元组 “five tuple”： (src_IP, dst_IP, src_port, dst_port, protocol)

FIGURE 1 Multipath TCP in the Stack



TCP连接建立过程

- 三次握手建立TCP连接
  - Create a socket to a single remote IP address/port



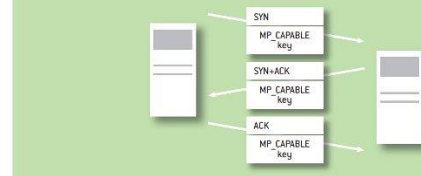
Each host tells its *Initial Sequence Number (ISN)* to the other host.

- 增加 subflows

协商 MPTCP 参数

- MPTCP的第一个子通道的建立遵守TCP的三次握手，唯一的区别是每次发送的报文段需要添加MP_CAPABLE的TCP选项和一个安全用途的key

FIGURE 2 Connection Establishment

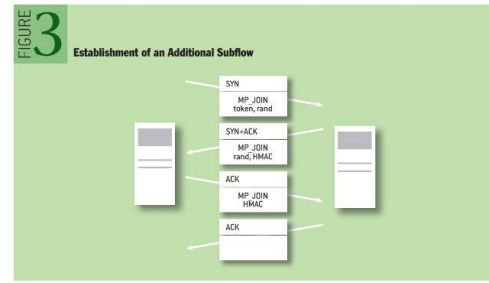


说明

- 打开一个会话，主机首先会向远端主机会发送一个 TCP SYN 消息，在 MPTCP 选项字段里包含了一个 MP_CAPABLE 信号
- 如果远端主机也支持 MPTCP，远端主机会返回一个 SYN + ACK 响应，同样在MPTCP 选项字段里包含了一个 MP_CAPABLE 信号。
- 会话使用了 ACK 和 MP_CAPABLE 信号完成 TCP 和 MTCP 握手过程，确保两端都得到了对方的 MPTCP 会话数据。
- 在整个会话过程中，两端交换了 64 位字节的会话密钥，同时各自生成一个 32位的哈希共享密钥。两个主机之间随后使用子链路的时候会用到这个共享密钥。

建立其他子通道

- 第二条子通道的建立依然遵守TCP的三次握手，而TCP选项换成了MP_JOIN。而token是基于key的一个hash值，rand为一个随机数，而HMAC是基于rand的一个hash值



说明

- MP_JOIN 包含接收端的哈希共享密钥和原始会话的 token 值，这样两端就能将新生成的 TCP 会话就能和原始会话关联起来了。MP_JOIN 还包含一个随机数，用来防止重放攻击。
- MP_JOIN 字段包含了发送端的地址，即使地址值被 NAT 转换了，两端还是能获得对方的原始地址。
- 会话两端能在任意端口生成 MP_JOIN 值。两端可以通过发送 ADD_ADDR 消息告知对方新的地址，同样可以通过发送 REMOVE_ADDR 删除地址。

数据接收和发送

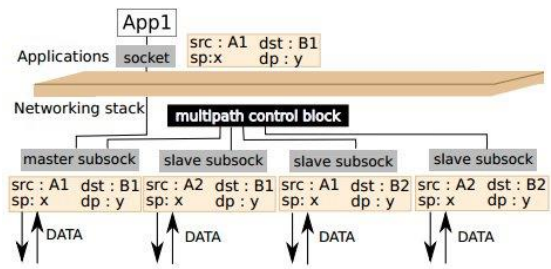
- MPTCP可以选择多条子通道中任意一条来发送数据
  - MPTCP如果使用传统的TCP的方式来发送数据，将会出现一部分包在一条子通道，而另一部分包在另外一条子通道。防火墙等中间设备将会收到TCP的序号跳跃的包，因此将会发生丢包等异常情况。
  - 为了解决这个问题，MPTCP通过增加DSN(data sequence number)来管理包的发送，DSN统计总的报文段序号，而每个子通道中的序号始终是连续。
- 接收包过程分为两个阶段
  - 每个子通道依据自身序号来重组报文段
  - MPTCP的控制模块依据DSN对所有子通道的报文段进行重组

拥塞控制

- MPTCP中拥塞控制的设计需遵守以下两个原则
  - MPTCP和传统TCP应该拥有相同的吞吐量，而不是MPTCP中每一条子通道和传统TCP具有相同的吞吐量。
  - MPTCP在选择子通道的时候应该选择拥塞情况更好的子通道。

MPTCP的实现

- master subsocket
  - 一个标准的sock结构体用于TCP通信
- Multi-path control bock(mpcb)
  - 提供开启或关闭子通道、选择发送数据的子通道以及重组报文段的功能
- slave subsocket
  - 对应用程序不可见，被mpcb管理并用于发送数据。



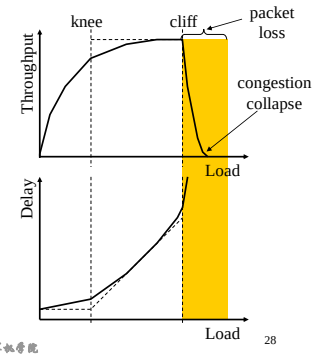
TCP拥塞控制

问题

- TCP拥塞控制机制的AIMD对应哪些阶段？如何计算拥塞窗口？
- 拥塞控制中，有哪些参数可以作为拥塞信号？适用范围是什么？
- 如果路由器不丢包，对分组延迟有什么影响？

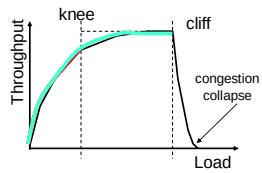
拥塞控制：View from a Single Flow

- Knee – point after which
  - 吞吐量缓慢增加
  - 时延迅速增加
- Cliff – point after which
  - 吞吐量开始迅速下降到0 (拥塞崩溃 congestion collapse)
  - 时延接近无限大



拥塞控制和拥塞避免

- 拥塞避免（Congestion avoidance）：Stay left of knee
- 拥塞控制（Congestion control）：Stay left of cliff
- 目标：
  - 有效利用网络资源
  - 网络资源分配的公平性
  - 防止或避免拥塞崩溃

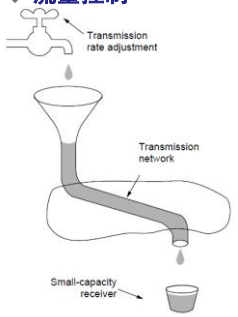


需要考虑的问题

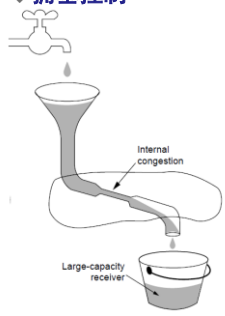
- 发送方如何知道发生拥塞?
  - 是否需要显式的网络反馈?
  - 基于网络性能进行推断?
- 发送方如何自适应?
  - 显式计算发送速率?
  - 端主机与其他主机进行协调?
  - 端主机是否需要全局信息进行决策?
- 性能目标是什么?
  - 最大化吞吐量?
  - 公平性（Fairness）?
  - 收敛性?

带宽资源分配：调整发送速率

◆ 流量控制



◆拥塞控制



带宽资源分配：调整发送速率

◆网络层可以使用不同的拥塞信号通知传输层端点调整发送速率

Protocol	Signal	Explicit?	Precise?
XCP	Rate to use	Yes	Yes
TCP with ECN	Congestion warning	Yes	No
FAST TCP	End-to-end delay	No	Yes
Compound TCP	Packet loss & end-to-end delay	No	Yes
CUBIC TCP	Packet loss	No	No
TCP	Packet loss	No	No

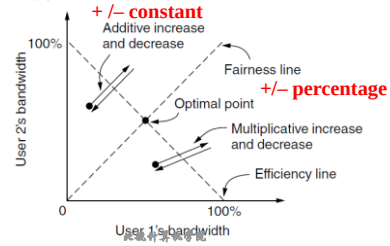
拥塞检测

◆ 显式信号
- 向源端返回特定分组 (e.g. ICMP Source Quench)
  - Control traffic congestion collapse
- 设置头部标志位 (e.g. ECN，IP的TOS字段)
- 部署问题: 需要路径上每个路由器支持
- 具有一定健壮性

◆ 间接信号
- 丢包 (e.g. TCP Tahoe, Reno, New Reno, SACK)
  - +relatively robust, -no avoidance
- 时延 (e.g. TCP Vegas)
  - +avoidance, -difficult to make robust
- 易于部署
- 健壮性问题？无线网络中如何处理？

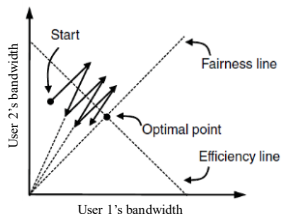
调整发送速率的方法

◆ 调整方法：加法增减；乘法增减
- Chiu和Jain，1989：二值拥塞反馈（loss or not）
- 如何收敛到兼顾公平性和有效性的最佳发送速率？
- 两个流竞争单条链路的带宽
  - 收敛性



调整发送速率的方法

- AIMD：加增乘减
  - The AIMD (Additive Increase Multiplicative Decrease) control law does converge to a fair and efficient point!
  - 收敛到兼顾公平和效率的最佳点
  - TCP



TCP拥塞控制的基本方法（复习）

- 慢启动 Slow-Start (SS)
- 拥塞避免 Congestion Avoidance (CA)
- 快速重传和快速恢复
- TCP的锯齿行为（ TCP Saw Tooth Behavior ）

TCP 协议特点

- 在端主机实现(end hosts)
  - 端到端理论（end-to-end arugment）
- 协议不断发展
  - 协议头部不变
  - 使用选项字段扩展
  - 在端节点改变处理过程
  - TCP的向后兼容特性

需要解决的端到端问题

运行：在整个Internet上

（1）滑动窗口协议：连接管理

（2）连接时延变化：自适应的RTT值估计

（3）分组重新排序

- 被延迟的分组对滑动窗口协议的影响
- 最大报文段生存期 MSL（Maximum Segment Lifetime）

（4）链路时延带宽乘积的影响

- 流量控制

（5）网络拥塞问题

Internet的拥塞现象

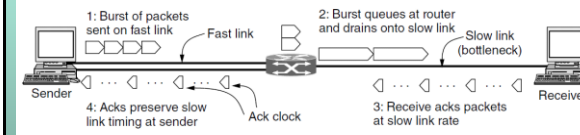
问题: 在1988年之前TCP是如何工作的?

拥塞崩溃 Congestion collapse:

- Breakdowns in performance noted in 1986 on NSFNet.
- 40Kb/s links operating as slow as 32b/s.
- NSFNet was a forerunner of today's Internet backbone (from 1986 to 1995).

TCP拥塞控制

- 拥塞窗口控制发送速率
  - 速率计算：cwnd / RTT
  - 控制：ACK clock (regular receipt of ACKs) paces traffic and smoothes out sender bursts



自同步：ACKs pace new segments into the network and smooth bursts

TCP拥塞控制模型

- 20世纪80年代后期，Van Jacobson引入TCP协议的拥塞控制机制
  - 在Internet出现拥塞现象
- 基于窗口的控制（window-based control）
  - 如何开始？—Slow start
  - 维护三个变量:
    - 拥塞窗口：cwnd
    - 接收方通告窗口：advertised window
    - 拥塞窗口阈值：ssthresh
- 通知发送方：
  - win = Min (advertised window, cwnd)

当检测到拥塞后，减小窗口

- 拥塞信号：Packet delay，Packet loss，mark packets
- 路由器拥塞指示

否则，增加窗口

TCP 拥塞窗口

- 每个TCP发送方维护一个拥塞窗口（ congestion window）
  - 允许传输的最大字节数
    - 通常为MSS（ maximum segment size ）的倍数
    - 至少为1个MSS长度
- 拥塞窗口的自适应
  - 丢失分组后，减小窗口：back off
  - 发送成功后增加窗口大小: optimistically exploring
  - 目的：发现合适的传输速率
- 特点
  - 优点: 避免网络中的显式反馈机制
  - 缺点: 速率估计的不稳定性：under-shooting and over-shooting the rate
    - 如何探测有效带宽（available bandwidth）？

TCP拥塞控制的基本思想

- **两个阶段:**
  - 慢启动 Slow-Start (SS)
    - “慢”：与TCP的最初行为相比
    - 两种情况下使用：连接开始阶段；丢包，计时器超时
    - 以指数方式有效增加拥塞窗口
  - 拥塞避免 Congestion Avoidance (CA)
    - 在慢启动达到拥塞窗口的阈值(ssthresh)后，进入CA阶段

慢启动（Slow Start）

- 目标：快速发现拥塞
- 方法：
  - 初始化： cwnd = 1 (slow start)
  - 每收到一个成功的ACK，增加拥塞窗口cwnd

    cwnd ← cnwd + 1
  - cwnd的指数增加

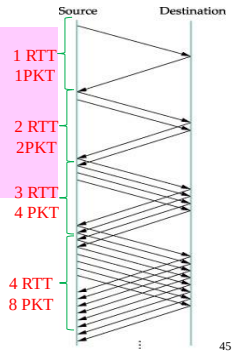
    each RTT: cwnd ← 2 × cwnd
  - 当 cwnd >= ssthresh，进入CA（拥塞避免）阶段
- 何时终止慢启动阶段？
  - Loss detected or...
  - cwnd > ssthresh
- 初始时，阈值ssthresh未知，如何试探？
  - Initially set to 64 KB
  - Will continue until a LOSS is detected.
  - ssthresh = cwnd / 2
  - Restart slow start

注意：*cwnd* increases exponentially

慢启动过程图示

慢启动期间传输的分组

第1个RTT, 发送1个分组；
第2个RTT, 发送2个分组；
第3个RTT, 发送4个分组；
。。。。



拥塞避免阶段

Congestion Avoidance

- 减慢 “Slow Start”：慢启动过程降速
- 线性增加
  - 如果*cwnd > ssthresh*，则有：
    each time a segment is acknowledged
    increment *cwnd* by *1/cwnd* (*cwnd += 1/cwnd*).
- 因此，只有当所有报文段被确认，有：

  cwnd ← *cwnd+1*

TCP 伪码（Pseudocode）

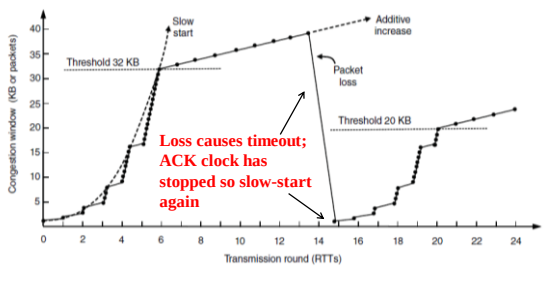
```
Initially:
    cwnd = 1;
    ssthresh = infinite;
New ack received:
    if (cwnd < ssthresh)
        /* Slow Start*/
        cwnd = cwnd + 1;
    else
        /* Congestion Avoidance */
        cwnd = cwnd + 1/cwnd;
Timeout:
    /* Multiplicative decrease */
    ssthresh = cwnd/2;
    cwnd = 1;
```


Additive Increase Multiplicative Decrease (AIMD)

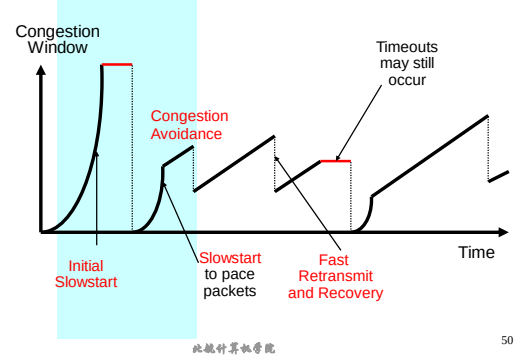
累次增加，成倍减少，（加增，乘减）

- Slow start during start up and after loss.
- Multiplicative decrease（MD）
  - 当分组丢失时（超时），将拥塞窗口大小降为一半
  ssthresh = cwnd / 2
- Additive increase（AI）
  - 数据发送成功（RTT内），线性增加窗口大小。
  每个RTT中，w← w+1

TCP Tahoe

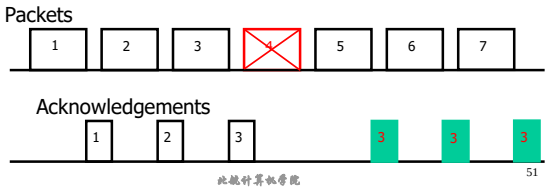


TCP Saw Tooth Behavior



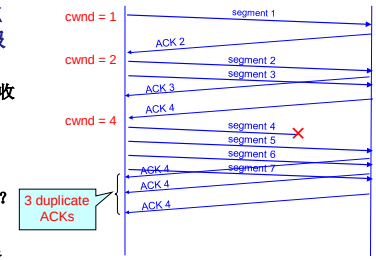
丢包问题（Packet Loss）

- 假设: 丢包指示拥塞的发生
- 丢包检测：
  - 重传计时器 Retransmission Time Outs (RTO timer)
  - 重复应答包 Duplicate ACKs (at least 3)



快速重传FR（Fast Retransmit）

- 在收到三个重复ACK后，重新发送一个报文段
  - 重复ACK表明接收方收到失序的报文段
- 注意:
  - 重复ACK产生的原因？
    - 分组失序
  - 若是小窗口，没有足够多的分组引起足够多的重复确认



快速重传FR（Fast Retransmit）

- 计数重复ack的数量
  - 引入新的状态变量: DUP_ACK
  - 若ACK值更新，该变量复位.
  - 否则，每收到一个重复ACK，加1.
- 若 DUP_ACK = 3
  - 修改拥塞窗口阈值
    - Set ssthresh ← max(cwnd/2, 2)
  - 进入慢启动
    - Set cwnd = 1
    - Set mode to SS（慢启动）

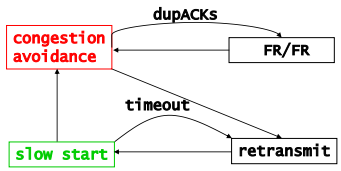
快速恢复FR（Fast recovery）

利用ACK同步分组的发送，不进入慢启动。

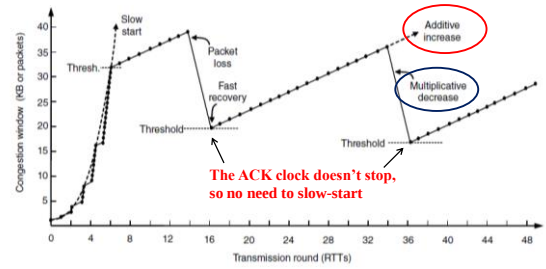
- 在快速重传之后：
  - *cwnd* ← *ssthresh/2*
    - I.e., don't reset *cwnd* to 1
  - 以原有速率进行发送，
    - Ack clocking rate is same as before loss
- 当RTO超时，进入慢启动： *cwnd* = 1
- 实现：
  - TCP Reno; most widely used version of TCP today

小结：SS/FR/FR

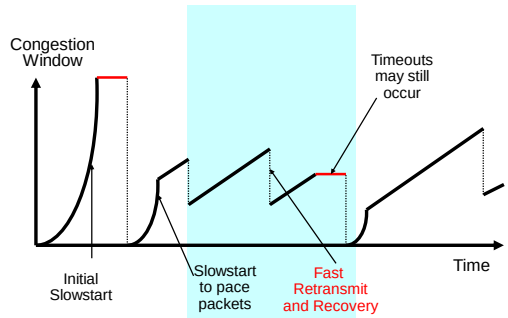
- 慢启动SS/快速重传FR/快速恢复FR
  - Fast recovery avoids slow start
  - dupACKs: fast retransmit + fast recovery
  - Timeout: fast retransmit + slow start



TCP Reno



TCP Saw Tooth Behavior

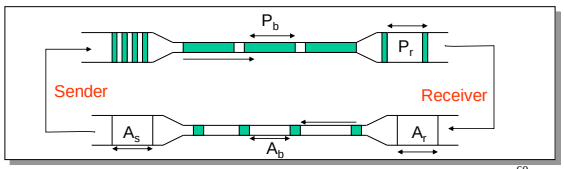


基于源端的拥塞避免机制

- 现象
  - 从网络中观察：路由器队列增加，可能导致拥塞发生
- 基本思想
  - 方法1：源端可以通过测量发送的连续分组中RTT的增加情况判断路由器分组队列增加。
    - 若分组时延增加，减小拥塞窗口（如1/8）
  - 方法2：根据RTT和窗口大小变化决定拥塞窗口调整的幅度
    - (CurrentWindow – OldWindow)×(CurrentRTT – OldRTT)
  - 方法3：估算吞吐量的变化进行调整。
  - 方法4：估算发送速率的改变量，将测量吞吐量变化率与理想吞吐量变化率做比较。

TCP 分组自同步

- 拥塞窗口对传输中分组的速率调整：Congestion window helps to "pace" the transmission of data packets
- 达到稳态：收到ACK,发送分组
  - 自同步行为：Self-clocking behavior



思考

- 端节点如何处理拥塞？
  - Uniform reaction to congestion – can different nodes do different things?
  - TCP friendliness, GAIMD, etc.
- 用排队时延作为拥塞指示？
  - TCP Vegas → BBR
- 非线性控制？
  - What about non-linear controls?
  - Binomial congestion control

其他拥塞控制方法：Vegas

- Vegas 算法试图在维持较好吞吐量的同时避免拥塞
- 它通过观察 RTT 来预测网络拥塞。
  - 当 RTT 增大时， Vegas 认为网络正在发生拥塞，于是线性降低发送速率。
- 利用 RTT 判断拥塞使得 Vegas 算法有较高的效率，但也导致采用 Vegas 的连接有较差的带宽竞争力。

其他拥塞控制方法：BIC-TCP

- BIC-TCP 算法的主要目的在于，即使在拥塞窗口非常大的情况下也能满足线性RTT公平性。
  - Binary Increase Congestion，BIC
- 使用二分查找增大（binary search increase）和最大探测（ max probing ）两种算法探测饱和点，通过最大值探测机制实现。相当于重新启动一个慢启动算法
- Linux 2.6.8 至 2.6.17 内核版本中默认开启该算法。

- BIC算法对窗口可能的最大值进行二分查找，它基于以下的事实：
  - 如果发生丢包的时候，窗口的大小是W1，那么要保持线路满载却不丢包，实际的窗口最大值应该在W1以下；
  - 如果检测到发生丢包，并且已经将窗口乘性减到了W2，那么实际的窗口值应该在W2以上。
  - 因此，在TCP快速恢复阶段过去之后，便开始在W2~W1这个区间内进行二分搜索，寻找窗口的实际最大值。于是定义W1为Wmax，定义W2为Wmin。
- 每收到一个ACK的时候，便将窗口设置到Wmax和Wmin的中点，一直持续到接近Wmax。

CUBIC

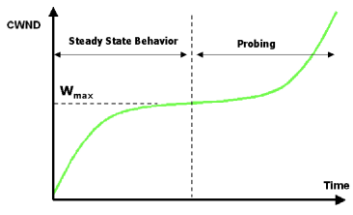
- 2008年提出CUBIC 算法改进了 BIC-TCP 算法中在某些情况下（低速网络）增长过快的不足，并对窗口增长机制进行了简化。
- 它通过一个三次函数来控制窗口的增长。
- 除此之外 CUBIC 支持TCP友好策略，确保在低速网络中CUBIC的友好性。
- 从Linux 2.6.18 内核版本开始 CUBIC 成为了 Linux 默认的 TCP 拥塞控制算法。

CUBIC

- Multiplicative decrease after loss
- W_{max} is window size before last loss

$$W_{cubic} = C(T-K)^3 + W_{max}$$

C is a scaling constant, and $K = \sqrt[3]{\frac{W_{max}\beta}{C}}$



BBR

- Google 在 2016 年下半年公开的一种开源拥塞控制算法，目前已经包含在了 Linux 4.9 内核版本中。

Cardwell N, Cheng Y, Gunn C S, et al. BBR: Congestion-based congestion control[J]. Queue, 2016, 14(5): 50.

- TCP BBR 已经在 Youtube 服务器和 Google 跨数据中心的内部广域网（B4）上部署。
- 解决两个问题
  - 有一定丢包率的网络链路上充分利用带宽
    - 适合高延迟、高带宽的网络链路
  - 降低网络链路上的 buffer 占有率，从而降低延迟
    - 适合慢速接入网络的用户

问题提出

- bufferbloat（缓冲区膨胀）问题
  - 当因为bottelneck buffers满而出现丢包时，会引起bufferbloat现象，网络延迟高；
  - 但是，当bottleneck buffers很小时，这时出现丢包，网络会误认为是发生了拥塞，从而降低发送窗口，这样就会造成吞吐量降低

TCP BBR的方法

- TCP BBR 不用丢包作为拥塞指示
- 分别估计带宽和延迟
  - 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。
- 在连接刚建立的时候，TCP BBR 采用类似标准 TCP 的慢启动，指数增长发送速率
- 根据收到的确认包，发现有效带宽不再增长时，就进入拥塞避免阶段。

（1）链路的错误丢包率只要不太高，对 BBR 没有影响；

（2）当发送速率增长到开始占用 buffer 的时候，有效带宽不再增长，BBR 就及时放弃了

TCP BBR的方法（续）

- 在慢启动过程中，由于 buffer 在前期几乎没被占用，延迟的最小值就是延迟的初始估计；慢启动结束时的最大有效带宽就是带宽的初始估计。
- 慢启动结束后，为了把多占用的 2 倍带宽 × 延迟消耗掉，BBR 将进入排空（drain）阶段，指数降低发送速率，此时 buffer 里的包就被慢慢排空，直到往返延迟不再降低。
- 排空阶段结束后，BBR 进入稳定运行状态，交替探测带宽和延迟。由于网络带宽的变化比延迟的变化更频繁，BBR 稳定状态的绝大多数时间处于带宽探测阶段。
- 带宽探测阶段是一个正反馈系统：定期尝试增加发包速率，如果收到确认的速率也增加了，就进一步增加发包速率。

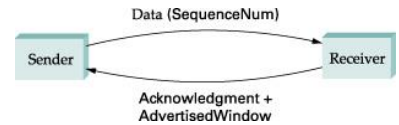
TCP协议相关

复习

复习：TCP连接管理

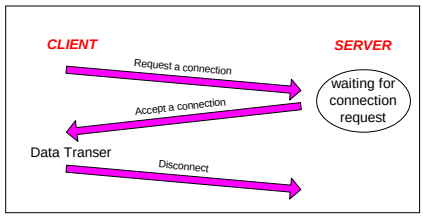
TCP连接标识

- 四元组
  - <SrcPort, SrcIPAddr, DstPort, DstIPAddr>
  - 注意：相同连接的不同实例
- TCP的连接过程示意图



面向连接

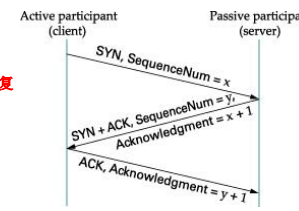
- 在数据传输之前，TCP建立连接（connection）：
  - One TCP entity is waiting for a connection ("server")：listening（）
  - The other TCP entity ("client") contacts the server：connect（）
- 连接建立过程复杂：可靠性
- 每个连接是全双工的（full duplex）



建立连接：参数协商

Three-Way handshake三次握手

- 通知对方发送时采用的初始序号
  - 为什么不从0开始？
  - 防止同一连接的两个实例过快重复使用同一序号
- 对序号进行应答
  - SYN-ACK: Acknowledge sequence number + 1
- 同时发送第二个SYN



- 为什么需要三次握手?（考虑连接出错的情况）
- 使用Wireshark分析TCP连接建立过程

Step 1: A's Initial SYN Packet

Flags: SYN FIN RST PSH URG ACK

A's port	B's port
A's Initial Sequence Number	
Acknowledgment	
20 | 0 | Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	

A 向 B 发起连接…

Step 2: B's SYN-ACK Packet

Flags: SYN FIN RST PSH URG ACK

B's port	A's port
B's Initial Sequence Number	
A's ISN plus 1	
20 | 0 | Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	

B tells A it accepts, and is ready to hear the next byte…
… upon receiving this packet, A can start sending data

Step 3: A's ACK of the SYN-ACK

Flags: SYN FIN RST PSH URG ACK

A's port	B's port
Sequence number	
B's ISN plus 1	
20 | 0 | Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	

A tells B it is okay to start sending
… upon receiving this packet, B can start sending data

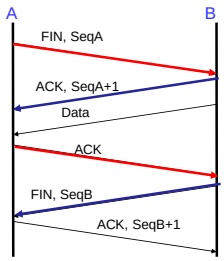
说明

- TCP服务器收到TCP SYN request包时，在发送TCP SYN + ACK包回客户机前，TCP服务器要先分配好一个数据区，缓存这个即将形成的TCP连接的信息。
- 一般把收到SYN包而还未收到ACK包时的连接状态称为半打开连接(Half-open Connection)。

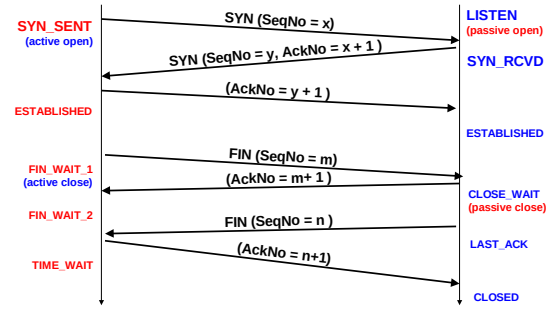
释放连接

两军队问题（two-army problem）

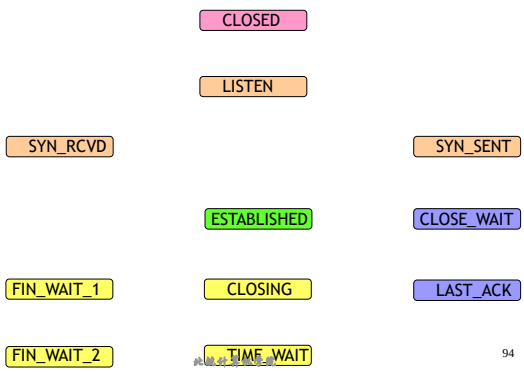
- 任何一方都可以发起释放连接
  - Send FIN signal
  - "I'm not going to send any more data"
- 另一方可以继续发送数据
  - Half open connection
  - Must continue to acknowledge
- 应答FIN
  - Acknowledge last sequence number + 1



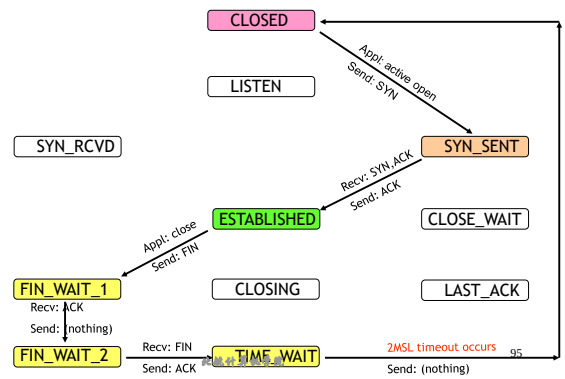
在“正常”连接期间TCP的状态



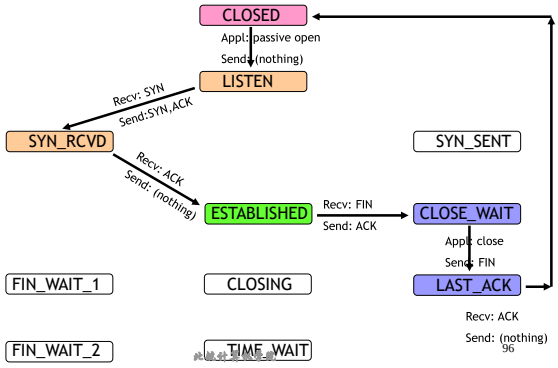
TCP 连接状态



TCP: 客户端打开和关闭连接



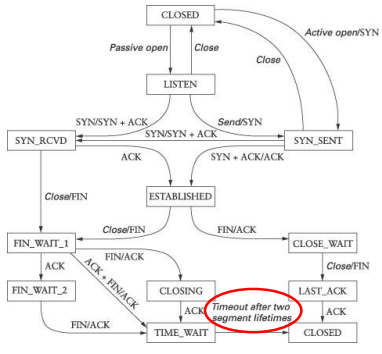
TCP: 服务器端打开和关闭连接



TCP 状态说明

State	Description
CLOSED	关闭连接
LISTEN	等待连接请求
SYN_RCVD	收到连接请求
SYN_SENT	发送连接请求
ESTABLISHED	连接建立，准备进行数据传输
CLOSE_WAIT	对端关闭连接
LAST_ACK	对端关闭连接，本地关闭，等待ACK
FIN_WAIT_1	本地关闭连接
FIN_WAIT_2	本地关闭连接，并且已经收到ACK'd
CLOSING	双方同时关闭连接
TIME_WAIT	等待网络丢弃有关分组

TCP状态转换图



TCP 状态转换图

- 有限状态机表达协议状态
  - 可以用矩阵表示
- 问题
  - 什么是状态转换（State transitions）？
    - 描述服务器在正常条件下采取的状态路径
    - 描述客户端在正常条件下采取的状态路径
    - 描述假设客户端首先关闭连接情况下采取的状态路径
  - 关于TIME_WAIT state
    - 该状态的目的是什么？
      - 保证至少连接的一方进入该状态
    - 双方如何进入关闭状态?

连接关闭的三种状态转换组合

- 主动关闭方（客户端）
  - ESTABLISHED → FIN_WAIT_1→FIN_WAIT_2→TIME_WAIT→CLOSED
- 被动关闭方（服务器端）
  - ESTABLISHED → CLOSE_WAIT→LAST_ACK→CLOSED
- 双方同时关闭
  - ESTABLISHED → FIN_WAIT_1→CLOSING→TIME_WAIT→CLOSED

2MSL Wait State

2MSL Wait State = TIME_WAIT

- 当TCP进行主动关闭（active close）时，发送最后的ACK，连接保持在 TIME_WAIT 状态的时间必须为2倍的MSL（maximum segment lifetime）.

  2MSL= 2 * Maximum Segment Lifetime

- 为什么?
  - 若对于FIN的ACK丢失，可能会导致一方重传FIN。若第二个FIN被网络延迟，产生副作用?
    - 考虑在同一个连接上另一对应用进程打开连接，延迟的FIN会使新的连接实例终止。
- MSL值通常设置为：
  - 2分钟

TCP 的选项

- TCP 头部
  - Ten mandatory fields
  - Optional extension field (usually during handshake)
- 例子
  - Maximum segment size (MSS)
  - Window scaling
  - Support for Selected ACKs
- 其他选项
  - Ignored by receiving host
- 路由器对TCP 选项字段的处理
  - Should ignore them, passing them through unchanged

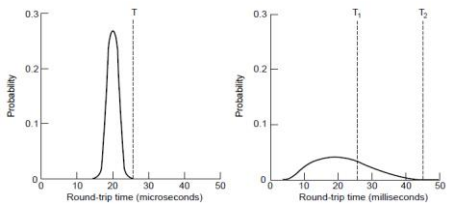
But, some middleboxes: (i) strip TCP options from some packets or (ii) drop packets with TCP options

问题

- 连接复位问题
  - 设置RST 标记进行连接复位
  - 何时设置RST标记？
    - 连接请求到达，但没有服务器进程在目的端口等待
    - 终止（异常中断）一个连接，导致接收方丢弃缓存的数据。
    - 接收方不应答RST报文段。
- 安全问题

TCP 计时器管理

◆ 数据链路层和传输层的时延（概率密度函数）



LAN case – small, regular RTT　　Internet case – large, varied RTT

TCP 的自适应重传

◆ 问题：如何准确估计RTT(round-trip time)?

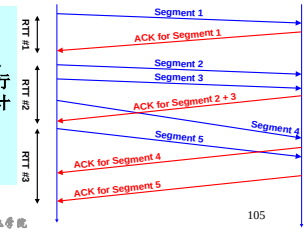
◆ 重传计时器(Retransmission Timer, RTO)

❖ Timeout value too small → results in unnecessary retransmissions

❖ Timeout value too large → long waiting time before a retransmission can be issued

RTT估算：动态算法

发送报文段和接收ACK的时间差.
问题：TCP并不对每个报文段进行应答，同时，每个连接只有一个计时器



RTT估计

◆ TCP 维持一个RTT的移动平均值，并将超时时间作为RTT的函数进行计算。

❖ Expect ACK to arrive after an "round-trip time"

❖ ... plus a fudge factor to account for queuing （平滑因子α）

◆ RTT简单估计：指数加权移动平均EWMA算法

❖ 通过观测ACK测量当前报文段的RTT：SampleRTT

❖ Smooth estimate: keep a running average of the RTT

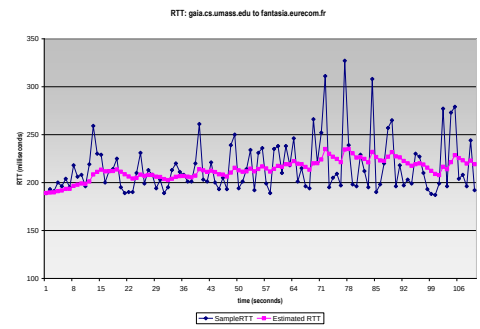
EstimatedRTT = α× EstimatedRTT + (1 –α) × SampleRTT

例如 平滑因子α= 0.875

❖ 超时计算:

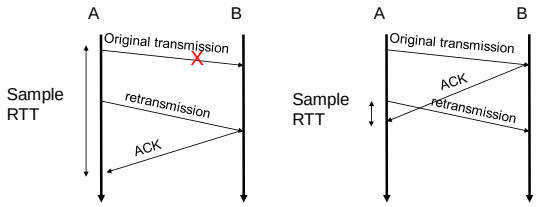
TimeOut = 2 ×EstimatedRTT

RTT估计



RTT简单估计的问题

- ACK：并非确认一次传输
- 当丢失分组，进行重传时：
  - 过高估计 Sample RTT（左图）
  - 过低估计 SampleRTT（右图）



TCP 自适应重传算法（Karn）

- 算法
  - 当TCP重传报文段时，停止计算RTT的样本值
  - 对于每次重传，
    - Double RTT estimate
    - Exponential backoff from congestion
- 存在问题
  - 样本变化的影响？
    - 样本变化小，RTT的估计值更为可信
    - 样本变化大，超时值可能远远大于估计值
  - 超时timeout 暗示拥塞
    - 如何解决拥塞？

TCP 自适应重传算法（Jacobson）

- 思想：考虑RTT测量值的变化。在高负载下，RTT值变化较大。
  - 计算RTT均值及其该均值的变化
- RTT测量值的平滑，同时考虑RTT变化：

Difference = SampleRTT – EstimatedRTT

EstimatedRTT = EstimatedRTT +（δ×Difference）

Deviation = Deviation + δ（| Difference |- Deviation ）

δ∈[0,1]

TimeOut = μ×Estimated + φ×Deviation

（μ=1，φ=4）

参见 [RFC2988，RFC6298] Computing TCP's Retransmission Timer

问题

- Linux采用的TCP超时重传时间是什么？
  - 可以通过抓包进行分析

参考：

RFC6298（Computing TCP's Retransmission Timer）的算法

TCP的计时器

- 重传计时器RTO
  - 最小值1秒
- 持续计时器 persistence timer
  - 避免死锁
  - 设置为重传时间的值，最大60秒
- 保活计时器 keepalive timer
  - 通常设置为2小时
- 连接终止计时器：TIMED WAIT
  - 最大报文段生命期的2倍：2 MSL =240秒
  - MSL 为 2分钟（RFC 793）

复习：TCP流量控制

Flow Control

TCP 的可靠传输机制

- 检测丢失数据: sequence number
  - Used to detect a gap in the stream of bytes
  - ... and for putting the data back in order
- 检测位差错：checksum
  - Used to detect corrupted data at the receiver
  - ...leading the receiver to drop the packet
- 从丢失数据中恢复：retransmission
  - Sender retransmits lost or corrupted data
  - Two main ways to detect lost packets
    - Retransmission timeout
    - fast retransmission

窗口：Receiver Window vs. Congestion Window

- 流量控制 Flow control
  - Keep a fast sender from overwhelming slow receiver
- 拥塞控制 Congestion control
  - Keep a set of senders from overloading the network
- 滑动窗口机制
  - TCP flow control: receiver window
  - TCP congestion control: congestion window
  - Sender TCP window =

    min { congestion window, receiver window }

TCP流量控制

- TCP使用滑动窗口协议
  - 在发送方，发送应答和设置窗口大小分离
  - 应答不自动增加窗口大小
- 缓冲区的大小不是唯一限制最大发送数据的因素
  - 每个ACK都包括接收方的通告窗口大小（*window advertisement*）
  - window = how many additional bytes (after last ACK'd byte) the receiver is prepared to accept
  - After this, the sender must stop and wait for an acknowledgment, even if buffer is available
    - Naturally, if buffer is unavailable then this dominates
- 允许窗口大小的调整
  - 从“window size”到接近“full window”
  - 对于TCP，传播时延（“propagation delay”）是通过整个网络的延迟

TCP的窗口管理

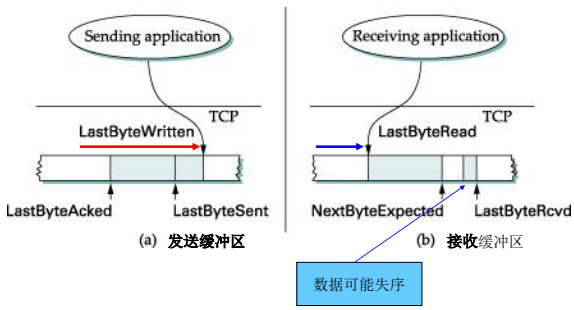
- 接收方向发送方返回两个参数

AckNo	window size (win)
32 bits	16 bits

- 含义:
  - 准备接收新的数据的序号为：
    SeqNo= AckNo, AckNo+1, ..., AckNo+Win-1
- 窗口变化
  - 接收方可以进行应答，而不打开窗口
  - 接收方可以改变窗口大小，而不应答数据

练习：用wireshark分析TCP报文

流量控制: 发送方和接收方缓冲区



流量控制：窗口变化讨论

- 接收方
  - LastByteRcvd - LastByteRead≤MaxRcvBuffer

  通告窗口大小：
  - AdvertisedWindow = MaxRcvBuffer - （（NextByteExpected - 1）- LastByteRead）
- 发送方
  - LastByteSent – LastByteAcked ≤ AdvertisedWindow

  有效窗口
  - EffectiveWindow = AdvertiseWindow - （LastByteSent – LastByteAcked）

问题：慢速接收进程如何使快速发送进程停止？

TCP 头部与ACK有关的字段

Flags: SYN
FIN
RST
PSH
URG
ACK

Source port	Destination port
Sequence number	
Acknowledgment	
HdrLen | 0 | Flags	Advertised window
Checksum	Urgent pointer
Options (variable)	
Data	

TCP发送ACK的规则

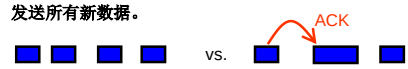
- 规则 1: Nagle's rule
  - *目的*: 减少小报文段的传输
  - *实现*：发送方不能发送多个负载为1字节的报文段 (例如，必须等待ACK)
- 规则 2: Delayed Acknowledgments
  - *目的*: 避免发送没有数据负载的ACK报文段
  - 避免通知小窗口：Silly window syndrome低能窗口综合症
  - *实现*：延迟发出一些ACKSs （合并ACK）

Nagle算法

- 动机
  - 交互式应用（如Telnet，rlogin）
    - 产生很多小分组 (e.g., keystrokes)
  - 小分组浪费带宽
    - 分组头部开销 (e.g., 40 bytes of header, 1 of data)
  - 希望降低小分组的数量
    - 能够强制每个分组的最小长度?
- 折中
  - 发送较大分组
  - ... 但不引入更多等待时延

Nagle算法：self-clocking方法

- 过程:
  - 若窗口大小允许，发送一个MSS大小的报文段
  - 如果有正在传输的报文段，
    - still awaiting the ACKs for previous packets
  - 否则，发送所有新数据。



- 说明：
  - ACK是激活定时器，一个RTT最多发送一个小分组
  - 性能影响：TCP连接的延迟
    - 交互式应用: enables batching of bytes
    - 块传输: transmits in MSS-sized packets anyway
- 关闭Nagle算法：应用程序设置TCP_NODELAY选项

Delayed ACK

- 接收方延迟发送 ACK
  - Upon receiving a packet, the host B sets a timer
    - Typically, 200 msec or 500 msec
  - If B's application generates data, go ahead and send
    - And piggyback the ACK bit
  - If the timer expires, send a (non-piggybacked) ACK
- 限制等待
  - Timer of 200 msec or 500 msec
  - ACK every other full-sized packet

性能考虑

- 高层应用可以控制窗口大小
  - Can change the socket buffer size from a default (e.g. 8-64Kbytes) to some maximum value
- 现代TCP版本 (linux, bsd, os x)支持自动调整
  - Historical source of performance problems on fast nets
- TCP头部的窗口大小（window size）字段限制的接收方能够通告的窗口
  - 16 bits → 64 KBytes
  - 10 msec RTT → 51 Mbit/second
  - 100 msec RTT → 5 Mbit/second （问题：网络效率分析）
- TCP扩展功能选项
  - Scaling factor： increases above 64KB limit