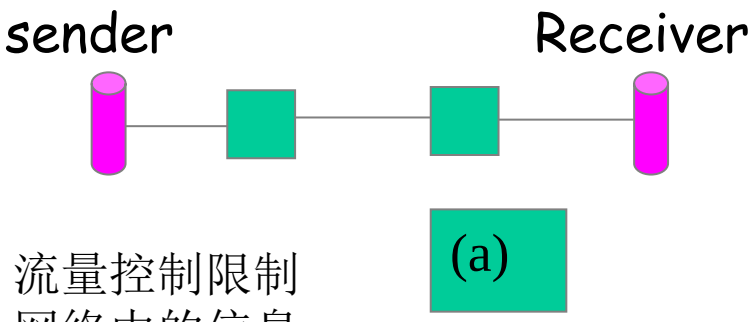
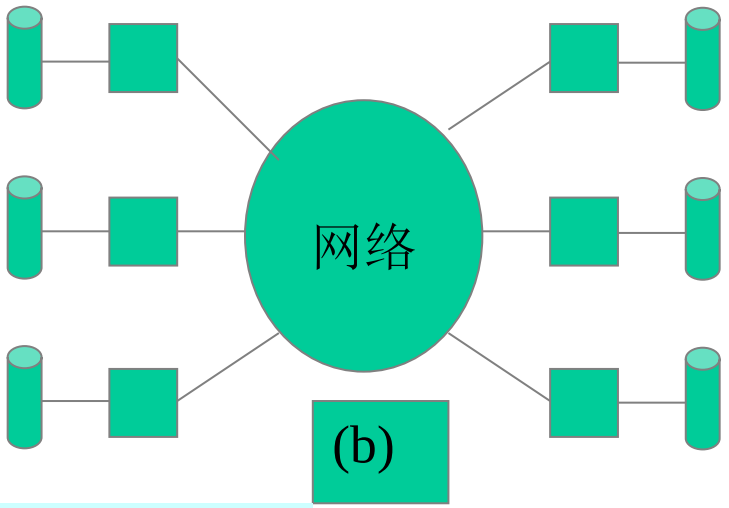
TCP

- 概述 √
- 报文格式 √
- 连接管理 √
- TCP的数据传输 √
- 流控与拥塞控制
- 错误控制 (Timer)

Congestion Control vs Flow Control

- Flow control is a mechanism to prevent a TCP sender from overwhelming a TCP receiver, congestion control is a mechanism to prevent a TCP sender from overwhelming the network.
- Congestion control is a global issue – involves every router and host within the subnet
- Flow control – scope is point-to-point; involves just sender and receiver.

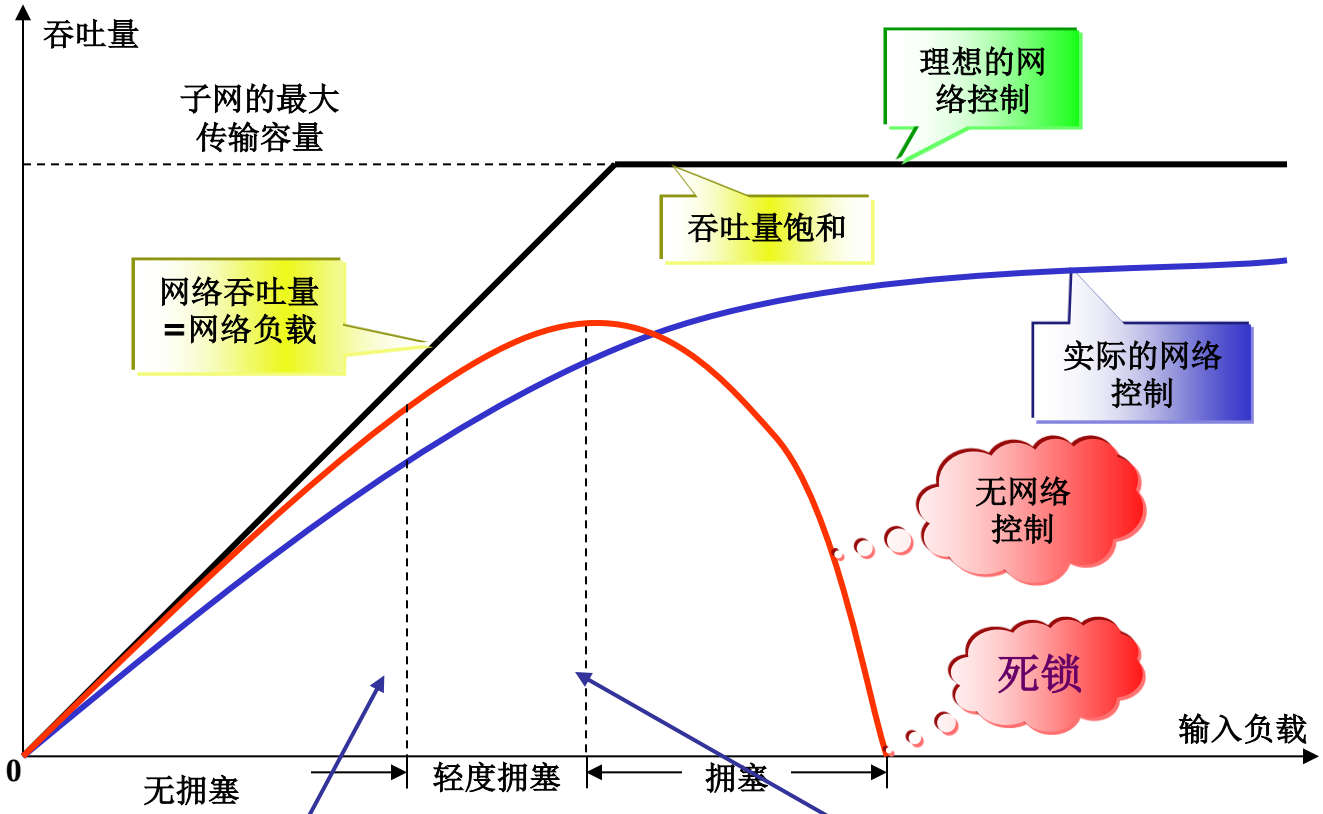




联系：流量控制限制了进入网络中的信息总量，可以在一定程度上减缓拥塞的作用。

流量控制和拥塞控制

拥塞控制与拥塞避免

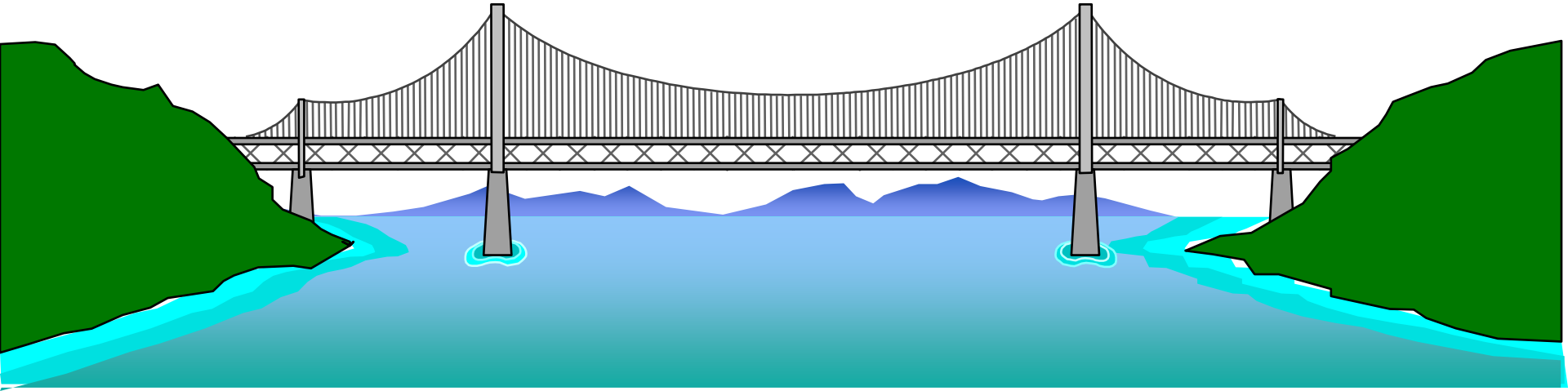


当网络负载继续增大到某一数值时，网络的吞吐量就下降为零，网络已无法工作.这就是所谓的"死锁"

拥塞避免：避免网络进入**轻度拥塞区域**

拥塞控制：控制网络不要进入**拥塞区域**

TCP流量控制



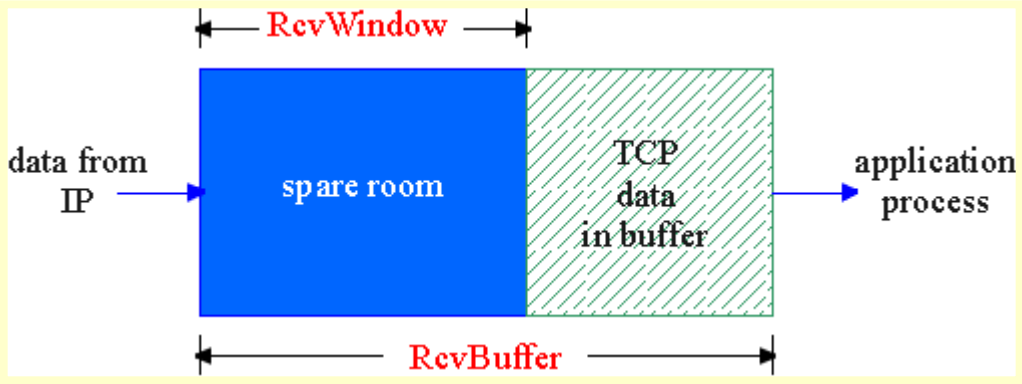
TCP流控

流控

发送方传输数据不能太多太快以致于淹没接收方

RcvBuffer = size or TCP Receive Buffer

RcvWindow = amount of spare room in Buffer



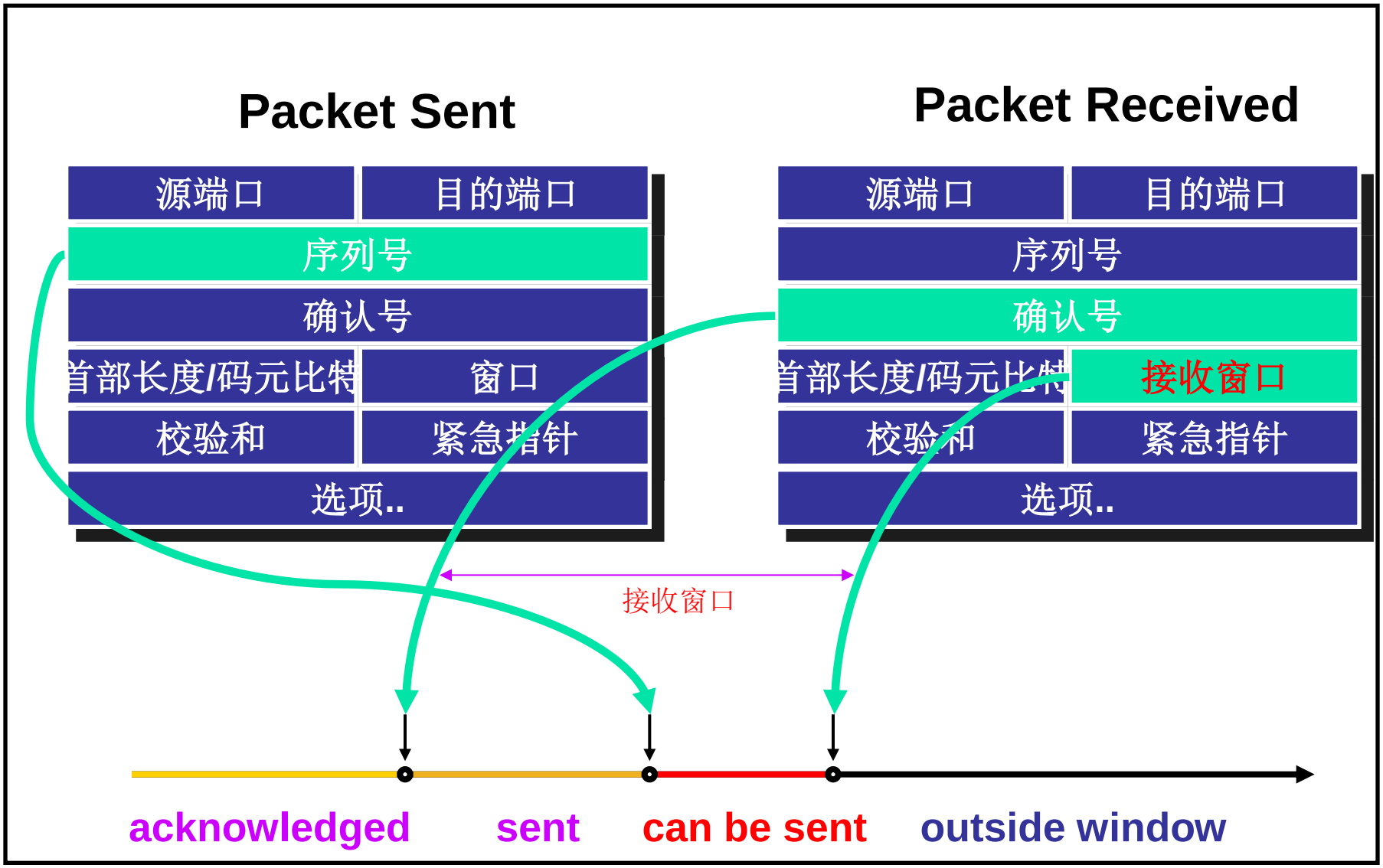
接收方缓存

接收方：明确告知发送方接收方可以接收的量

- TCP报头中的接收窗口域

发送方：保证发送的数据不超过接收窗口值

滑动窗口流量控制：发送方



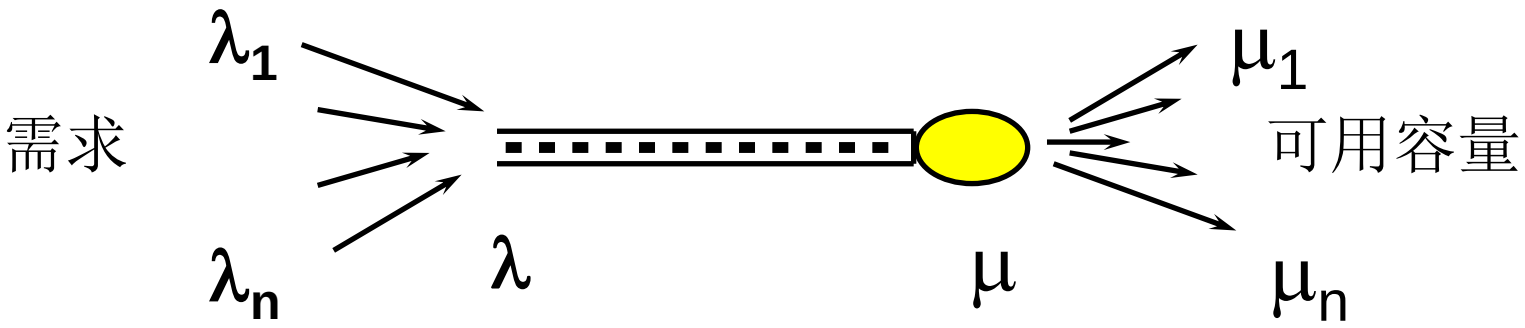
利用可变接收窗口进行流量控制举例

主机A向主机B发送数据，双方商定的窗口值是400，每一个报文段为100字节，序号的初始值为1。主机B进行了3次流量控制。

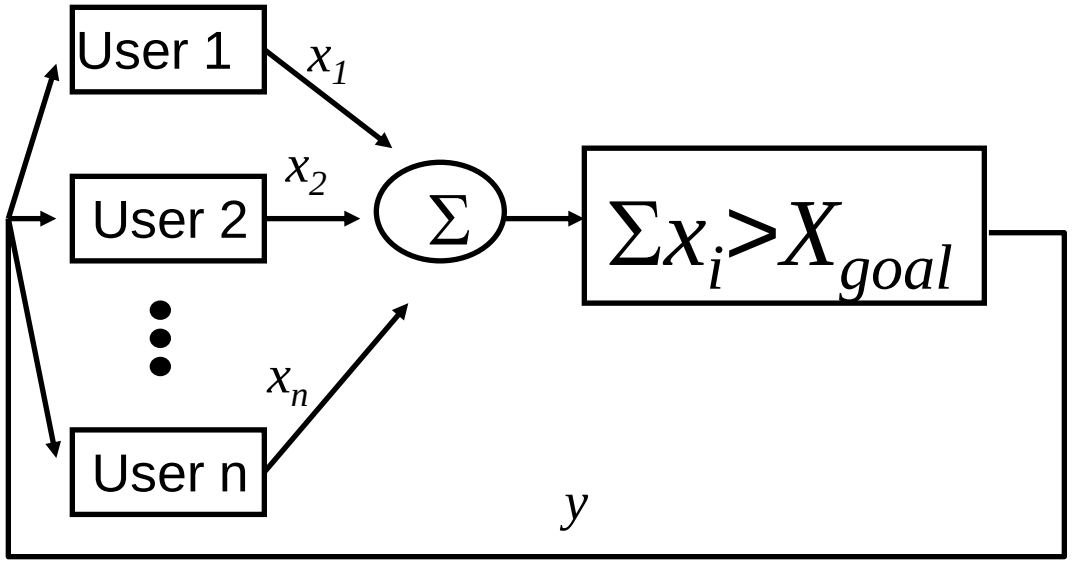
TCP拥塞控制

拥塞问题

问题: 对资源的需求超过了可用资源（$\mu < \lambda$)



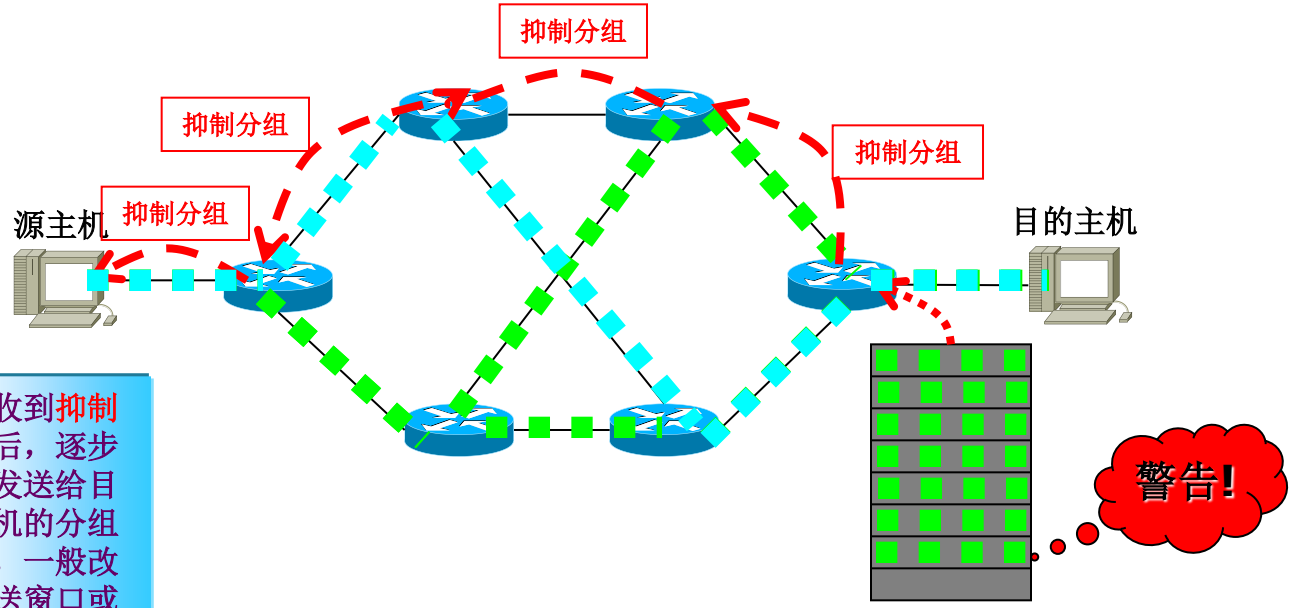
解决方法: 避免用户的发送量超过网络的能力



拥塞控制的基本方法

- 流量监管与整形。把主机向网络输出数据的速率调整到一个固定的平均速率，以使路由器能够以现有资源平稳地处理它。流量监管与整形的两个代表算法是漏桶算法(leaky bucket algorithm)和令牌桶算法(token bucket algorithm)。
- 拥塞预警法。其原理是：路由器建立一种拥塞预警机制，当轻度拥塞发生时，通知源主机减少发送数据的数量，避免拥塞加剧。ICMP(internet control message protocol)协议的源抑制报文就采用拥塞预警原理。
- 基于路由器的主动队列管理(AQM)。
- TCP implements host-based, feedback-based, window-based congestion control.

ICMP Choke Packets

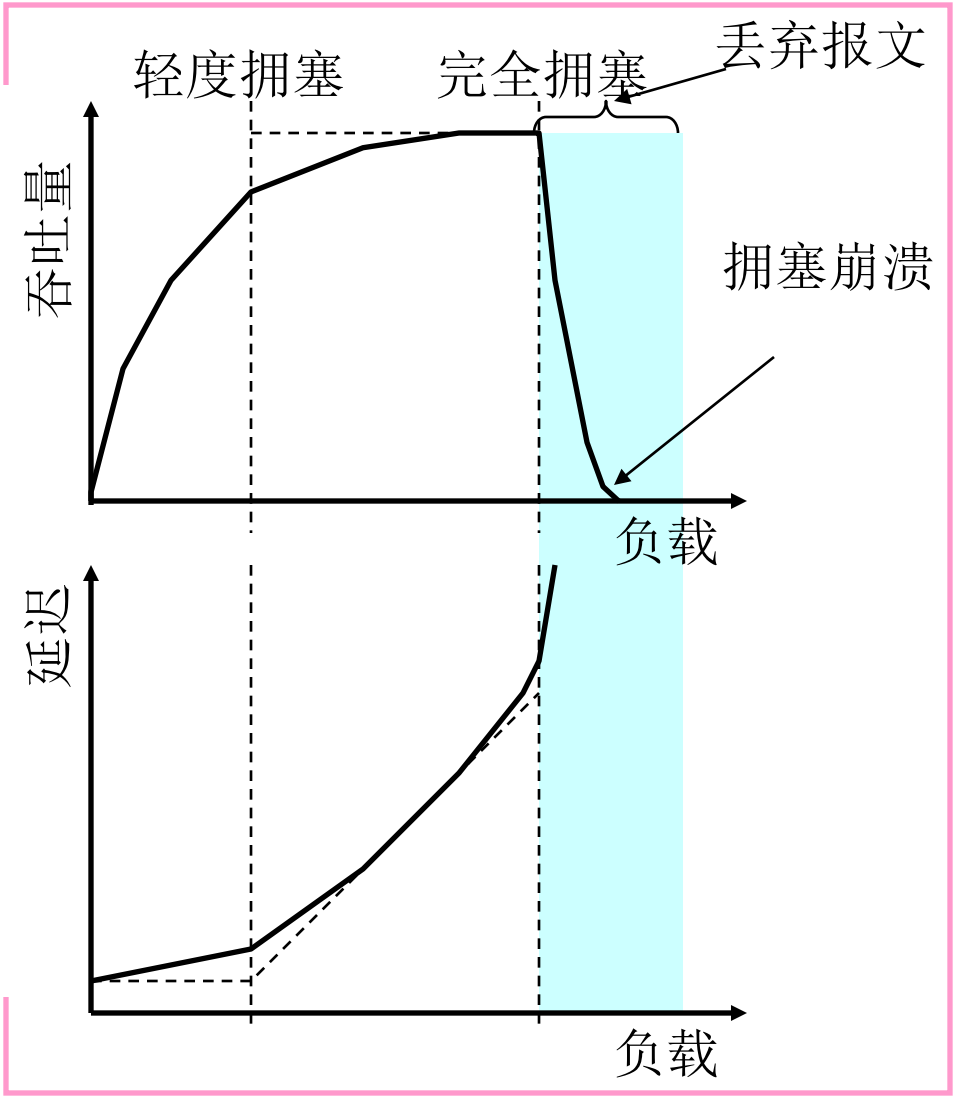


主机收到抑制分组后，逐步减少发送给目的主机的分组数量，一般改变发送窗口或采用漏桶输出率等。

图　基于源抑制的拥塞控制策略

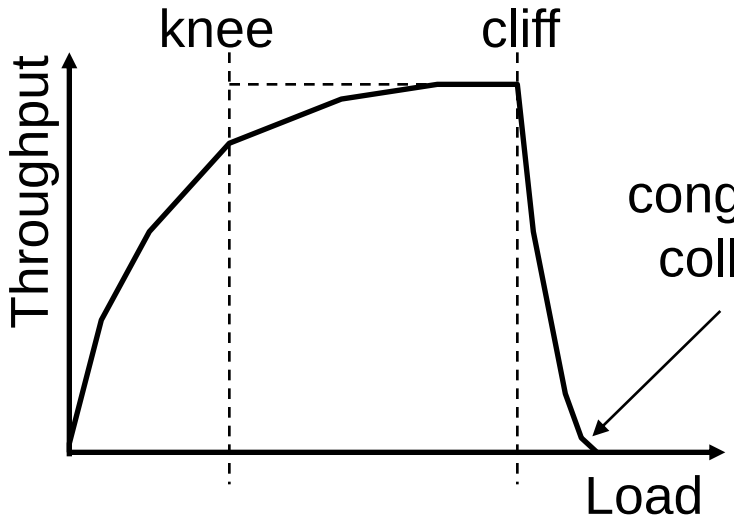
拥塞: A Close-up View

- 轻度拥塞 – 该点后
  - 吞吐量增长缓慢
  - 延迟增加加快
- 完全拥塞 – 该点后
  - 吞吐量快速下降直至零 (拥塞崩溃)
  - 无穷大延迟
- 对于M/M/1排队模型
  - 延时 = 1/(1 – 利用率)



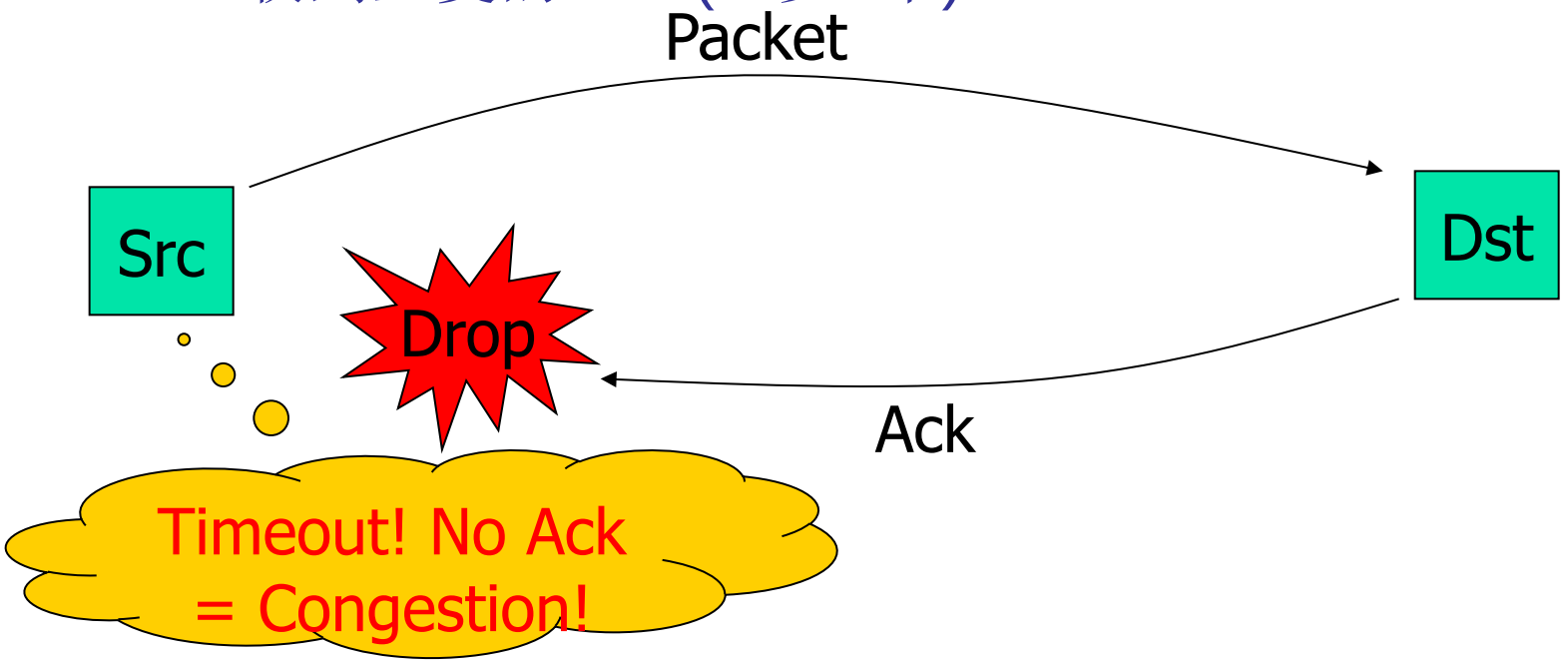
拥塞控制与拥塞避免

- 阻塞控制的目标
  - 控制在悬崖（cliff）的左边
- 阻塞避免的目标
  - 控制在拐点（knee）的左边
- 悬崖的右边:
  - 由于拥塞导致崩溃

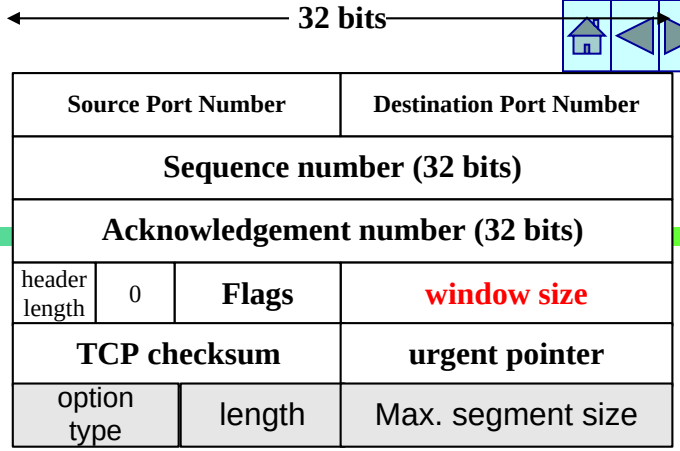


拥塞检测

- 发生报文丢失时表示发生了拥塞，TCP 发送方通过以下方式检测丢包
  - ✓ 传输定时器超时（丢包）
  - ✓ 收到重复的ACK(至少3个)



TCP 拥塞控制



- TCP 拥塞控制机制. 由发送方实现
- 发送方的发送窗口大小设置如下:

发送窗口 = MIN (流控窗口, 拥塞窗口)

其中

- 流控窗口rwnd：由接收方通报(即TCP报头中的窗口字段的值)，表示接收方最多可以接收的数据量
- 拥塞窗口cwnd：由发送端根据网络的反馈来动态调整
- TCP协议同时具有流控和拥塞控制功能：rwnd prevents the sender from overwhelming the receiver and cwnd prevents the data from overwhelming the network.

TCP congestion control

- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (**`Congwin`** as large as possible) without loss
  - *increase* **`Congwin`** until loss (congestion)
  - loss: *decrease* **`Congwin`**, then begin probing (increasing) again

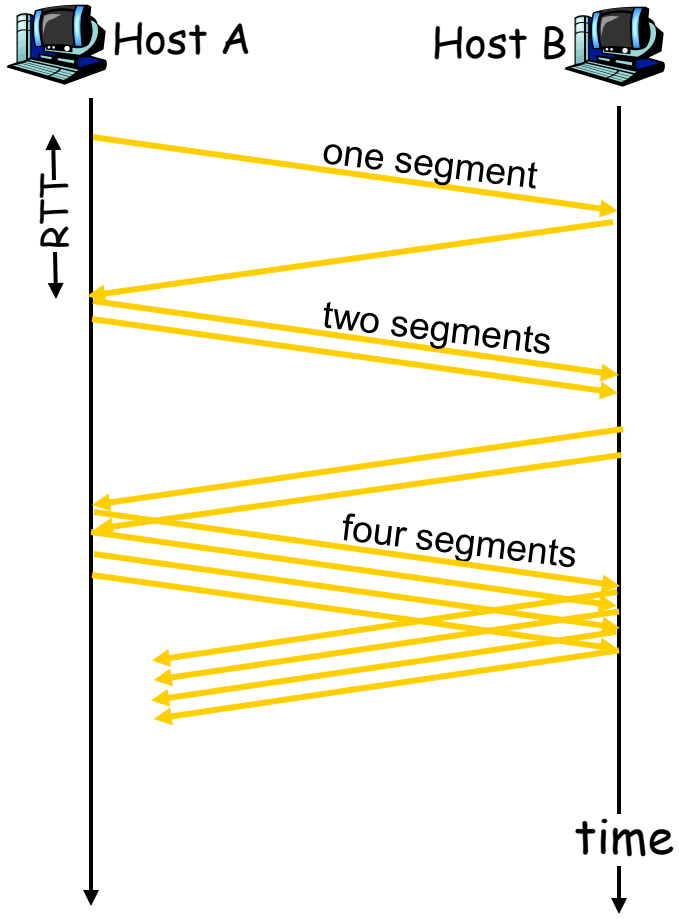
- two "phases"
  - slow start
  - congestion avoidance
- important variables:
  - **`Congwin`**
  - **`threshold`**: defines threshold between two slow start phase, congestion control phase

TCP Slowstart

Slowstart algorithm

initialize: Congwin = 1
for (each segment ACKed)
    Congwin++
until (CongWin > threshold
OR congestion event)

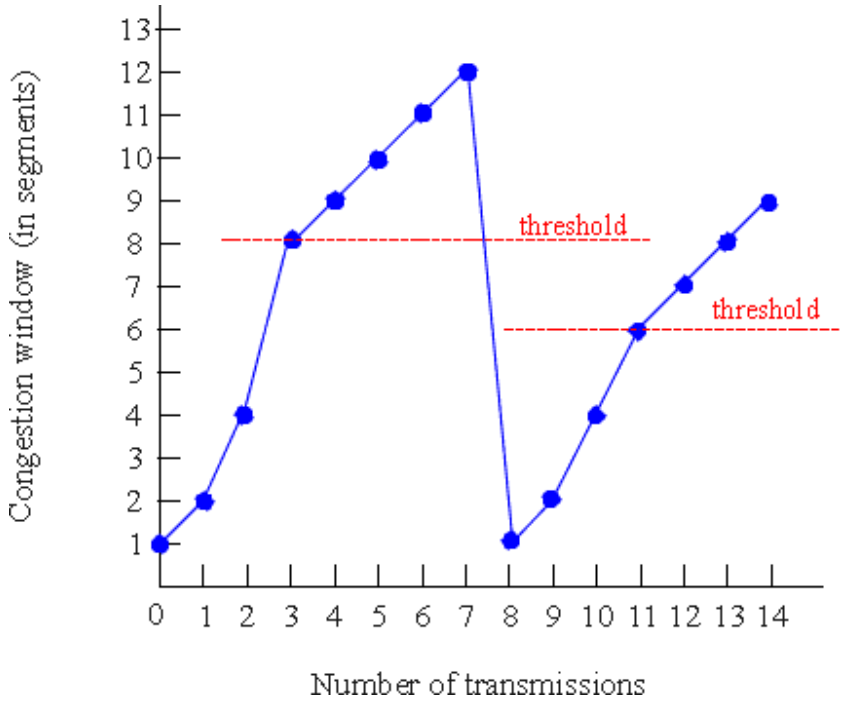
- exponential increase (per RTT) in window size (not so slow!)
- In case of timeout:
  - Threshold=CongWin/2



TCP Congestion Avoidance

Congestion avoidance

```
/* slowstart is over          */
/* Congwin > threshold */
Until (timeout) { /* loss event */
  every ACK:
      Congwin += 1
  }
threshold = Congwin/2
Congwin = 1
perform slowstart
```



TCP Taheo

拥塞响应

- TCP 采用不同的算法来处理阻塞问题
  - Congestion = timeout

    TCP Tahoe
  - Congestion = timeout || 3 duplicate ACK

    TCP Reno -Tahoe + 快速恢复

    TCP New Reno – 改进的 Reno
  - Congestion = higher latency

    TCP Vegas
- 其它:
  - SACK TCP
  - Fack TCP

TCP Reno

- TCP Tahoe的问题：

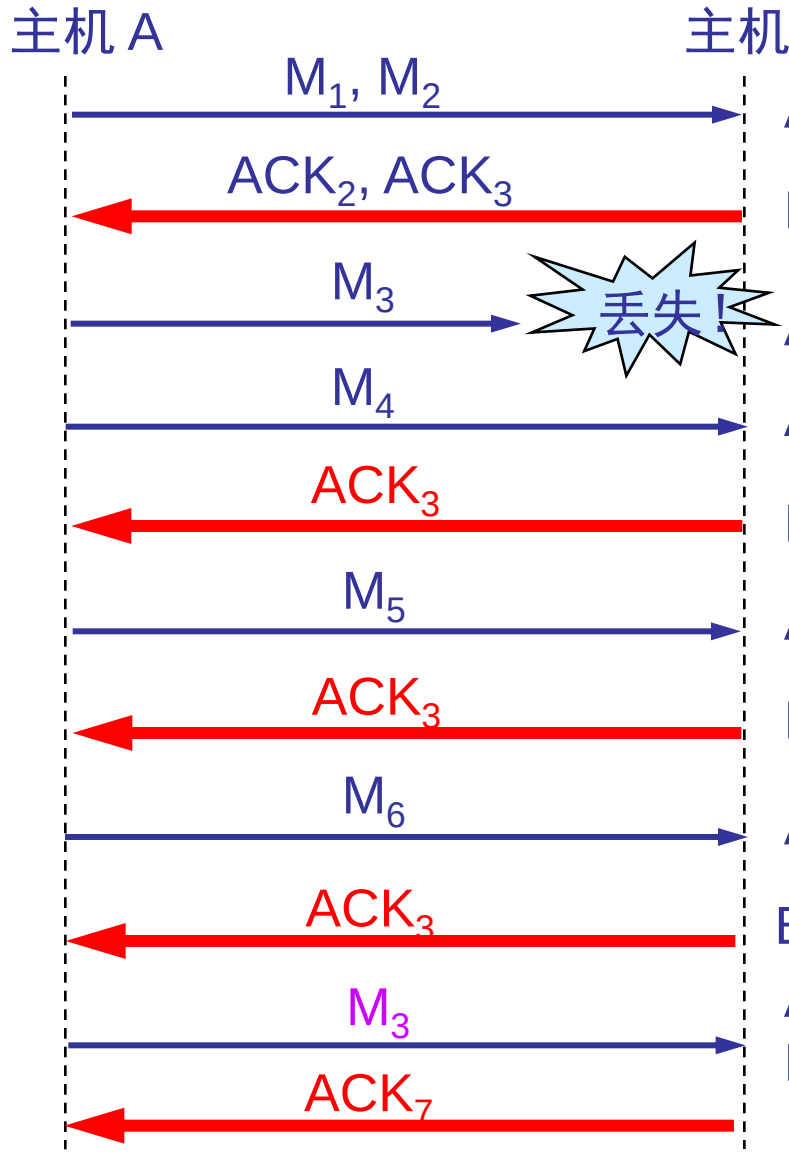
(1)如果一个报文分组丢失了，直到超时才能被发现----时延大；

(2)当重传定时器(RTO)超时，cwnd被重置为1，重新进入慢启动阶段，这会导致过大地减小发送窗口尺寸，降低TCP连接的吞吐量。

TCP Reno

- Fast retransmit:
  - After receiving 3 duplicate ACK
  - Resend first packet in window.
    - Try to avoid waiting for timeout
- Fast recovery:
  - After retransmission do not enter slowstart.
  - Threshold = Congwin/2
  - Congwin = 3 + Congwin/2
  - Each duplicate ACK received Congwin++
  - After new ACK
    - Congwin = Threshold
    - return to congestion avoidance

快速重传举例



A 发送 M_1 和 M_2

B 确认 M_1 和 M_2

A 发送 M_3 但丢失了

A 发送 M_4

B 只能再次确认 M_2(因为 M_3 没有收到）

A 发送 M_5

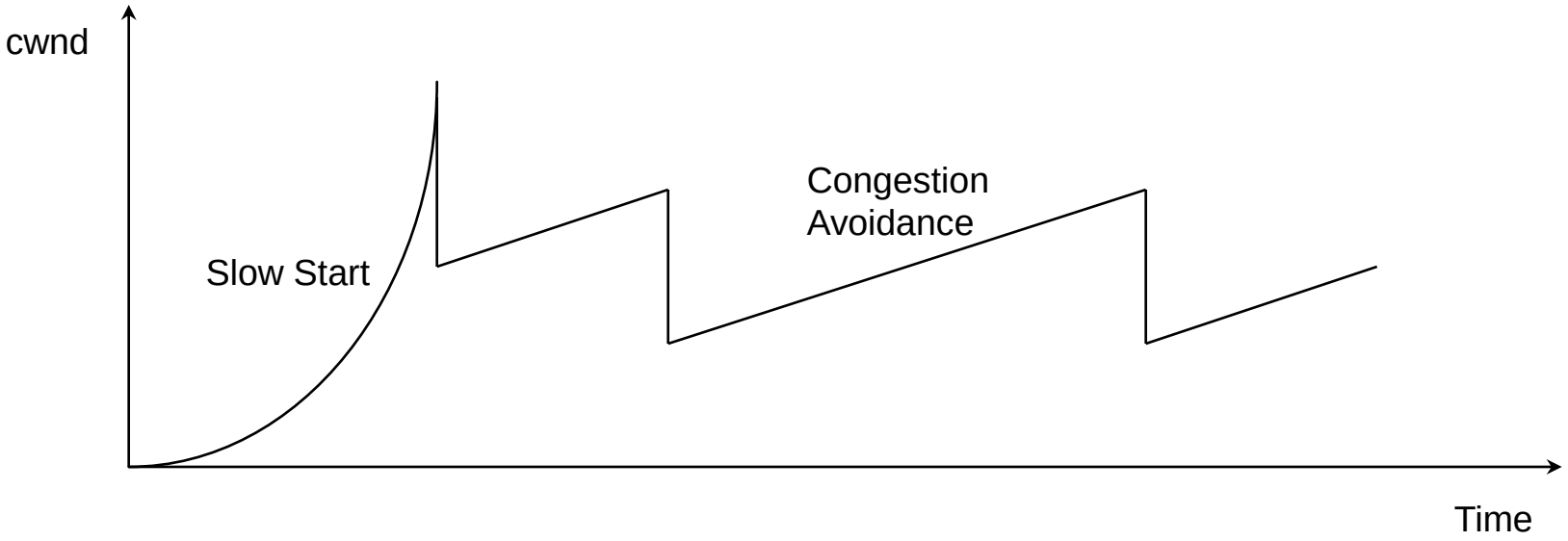
B 发送第二个重复确认 ACK_3

A 发送 M_6

B 发送第三个重复确认 ACK_3

A 收到了三个重复的确认 ACK_3，就立即重传 M_3，而不必等待超时再重传。

快速重传与快速恢复



- 三个重复的ACK后，开始快速重传
  - 避免了代价高昂的超时
- 不必再次慢启动
- 在稳定状态，*cwnd* 在最优窗口大小附近振荡

Reno TCP存在的问题

- RenoTCP算法仍有不足:

(1)在检测到拥塞后，发送方要重传从数据包丢失到检测到丢失这段时间内发送的全部数据包(GO-BACK-N)，而这中间有些数据包可能已经正确传送到接收方，不必重传。

(2)S. Floyd和 v. Jacobson通过NS模拟证明了在发生多个报文段丢失的情况下，Reno TCP不能够进行有效的快速恢复。

- 针对上述问题，近年来又出现了许多新的TCP版本。

New Reno

- 与 Reno TCP 类似，但对Reno TCP中的“快速恢复”算法进行了修正，它考虑了一个发送窗口内多个数据包丢失的情况。
- 在Reno 的“快速恢复”算法中，发送方收到一个不重复的应答后就退出“快速恢复”状态。而在 NewReno 中，每个往返时间重传一个丢失的报文段，直至所有的丢包全被重传完，在收到对新数据的确认后才退出快速恢复阶段。

SACK(Selected Acknowledgement)

- TCP SACK (Selected Acknowledgement)
  - TCP (Thaoe) sender can only know about a single lost per RTT
  - SACK option provides better recovery from multiple losses
    - The sender can transmit all lost packets
  - Operation
    - Add SACK option into TCP header
    - The receiver sends back SACK to sender to inform the reception of the packet
    - Then, the sender can retransmit only the missing packet

TCP Tahoe/Reno存在的问题

- 事后动作，反应太慢----AQM
- 以报文丢失作为拥塞的唯一判断依据----对于无线网等场合不准确
- 单比特的拥塞指示（拥塞/不拥塞）----粒度太粗，不能反映拥塞的程度
- 采用AIDM的窗口更改策略 ----不适合于高速长距离(大RTT)的场合
- ……

The Problem of TCP in high speed network

- **Congestion Control for High-Bandwidth-Delay-Product Networks**
- Sustaining high congestion windows:
- A Standard TCP connection with:
  - ✓ 1500-byte packets;
  - ✓ a 100 ms round-trip time （RTT）;
  - ✓ a steady-state throughput of 10 Gbps;
- would require:
  - ✓ an average congestion window of 83,333 segments;
  - ✓ and at most one drop (or mark) every 5,000,000,000 packets

(or equivalently, at most one drop every 1 2/3 (1.6) hours)

This is not realistic.

New Advances in TCP Congestion Control

- **TCP Vegas**
- **TCP Westwood (RCPW)**
  在TCP发送端通过检测返回的ACKs速率来持续测量网络的有效带宽，在发送端自适应地修改拥塞窗口
- **HighSpeed TCP (HSTCP)**
  2003年,Floyd S.
  针对高速链路修正的TCP拥塞控制新算法
- **Fast TCP**
  2002/2003年 针对高速网络的一个新的TCP拥塞控制算法
  基于优化理论，发送窗口不是线性增长而是以分段速率递进
- **XCP (eXplicit Control Protocol)**
  针对高带宽时延乘积网络
- **Highspeed TCP Low priority**
- **Binary Increase Control TCP (BIC TCP)**
- **Hamilton TCP (H TCP)**
- **Compound-TCP**
- **SLAC TCP (美国斯坦福线性加速器中心（*SLAC*）)**
- **Parallel TCP Reno**
- **Scalable TCP**
- ……

TCP Vegas

- Uses congestion avoidance instead of congestion control
  - Reno: Congestion control React to congestion *after* it occurs
  - Vegas: Congestion avoidance Predict and avoid congestion *before* it occurs

TCP Vegas

- Idea: track the RTT
  - Try to avoid packet loss
  - latency increases: lower rate
  - latency very low: increase rate
- Implementation
  - sample_RTT: current RTT
  - Base_RTT: min. over sample_RTT
  - Expected = Congwin / Base_RTT
  - Actual = number of packets sent / sample_RTT
  - Δ =Expected - Actual

TCP Vegas

- Δ = Expected - Actual
- Congestion Avoidance
  - two parameters: α and β, $\alpha<\beta$
  - If ($\Delta < \alpha$) Congwin = Congwin +1
  - If ($\Delta > \beta$) Congwin = Congwin -1
  - Otherwise no change
  - Note: Once per RTT
- Slowstart
  - parameter γ
  - If ($\Delta > \gamma$) then move to congestion avoidance

For More Information

- FAST TCP
  - http://netlab.caltech.edu/FAST/
- Scalable TCP
  - http://www-lce.eng.cam.ac.uk/~ctk21/scalable/
- Highspeed TCP
  - http://www.icir.org/floyd/hstcp.html
- Highspeed TCP Low-Priority
  - http://dsd.lbl.gov/DIDC/PFLDnet2004/papers/Kuzmanovic.pdf
  - *TCP*-LP's Homepage

    http://www.ece.rice.edu/networks/TCP-LP/
- Binary Increase Control (BIC) TCP
  - http://www.csc.ncsu.edu/faculty/rhee/export/bitcp/
  - http://netsrv.csc.ncsu.edu/twiki/bin/view/Main/BIC

Even More Information

- XCP:
  - http://www.acm.org/sigcomm/sigcomm2002/papers/xcp.html
- Hamilton TCP
  - http://www.hamilton.ie/net/main.htm?tcp
- QuickStart:
  - http://www.icir.org/oyd/papers/draft-amit-quick-start-01.txt
- SLAC TCP (美国斯坦福线性加速器中心（*SLAC*）)
  - http://www-iepm.slac.stanford.edu/monitoring/bulk/fast/
- Internet2 (Stanislav Shalunov)
  - http:// www.internet2.edu/~shalunov/
- TransPAC
  - http://www.transpac.org
- APAN NOC
  - http://www.jp.apan.net/noc/

TCP

- 概述 √
- 报文格式 √
- 连接管理 √
- TCP层的数据传输 √
- 流控和拥塞控制 √
- 错误控制 (Timer)

TCP错误控制

- TCP 实现了回退N帧重传机制
- TCP 为每个连接维持了四个定时器
- TCP 同时进行错误控制和拥塞控制 (TCP 假定错误是由拥塞引起的)
- TCP 允许加速重发 (*快速重传*)

TCP的定时管理

- TCP使用4个定时器来完成各种功能：
  - **重传**定时器 (retransmission timer)
    - 发送每个段的同时启动此定时器。在定时器超时前，收到该段确认，关闭此定时器。在超时前，没有收到该段确认，重发该段。
    - 问题：超时间隔设为多长合理？
  - **持续**定时器 (persistence timer)
    - 用于解决死锁。在发送接收双方由于某种原因（例如丢包）而相互等待，产生死锁时，如果持续定时器超时，发送方向接收方发送一个探测报文，打破死锁。
  - **keepalive** timer
    - 当一个连接被长时间闲置时，如果keepalive定时器超时，就发送一个报文去检测连接的另一方是否依旧存在。如果没有得到相应，就中止该连接。
  - **TIMED WAIT** timer
    - 在释放连接的操作中使用此TIMED WAIT 定时器。时间长度为分组TTL的两倍，以确保当一个连接断开之后，所有被此连接创建的分组都完全消失。

TCP 重发定时器

- 重发定时器的设定对效率至关重要
  - ✓ 超时值过小 -> 导致不必要的重传
  - ✓ 超时值过大 -> 导致过长的等待时间
- 问题在于网络中的延迟不是固定的
- 因此, 重传定时器必须能够自适应

Karn 算法

无重传时，TCP根据式（1），（2）确定重传时间

$TIMEOUT=\beta*RTT$ （1）

其中，$\beta>1$的常数（TCP推荐$\beta=2$）

RTT：估算的往返时间，是一个加权平均值。

$RTT=\alpha*Old\_RTT+(1-\alpha)*New\_Round\_Trip\_Sample$ (2)

其中，$0\leq\alpha\leq1$ 常数加权因子

Old_RTT：上一个往返时间估算值。

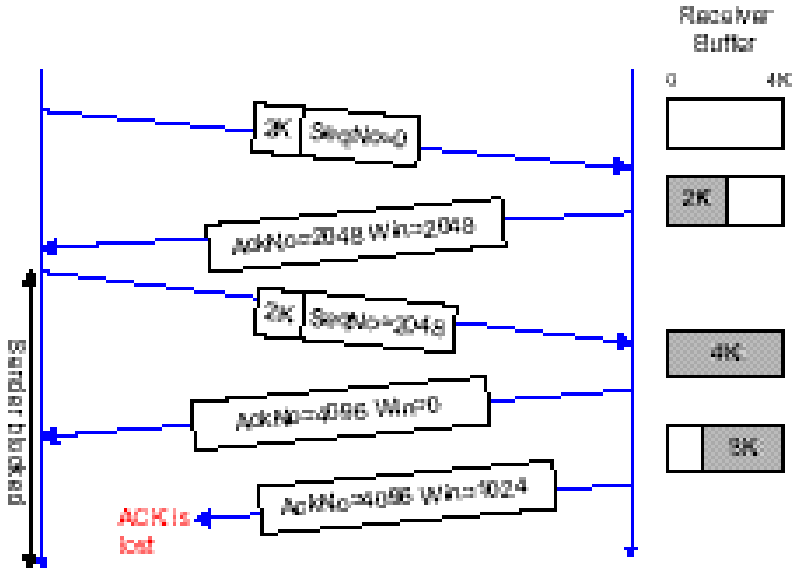
New_Round_Trip_Sample：实际测出的前一个段的往返时间（样本）。

当发生重传时，用下列公式计算新的重传时间

$New\_Timeout=\gamma*Timeout$ (3)

其中，$\gamma>1$，典型值为$\gamma=2$，即每重传一次，超时值加倍。

TCP 持续定时器



持续定时器:

促使发送方周期性地询问接收方接收窗口的大小 (即，接收窗口大小探测)

- 假定窗口大小变小为零后，启动窗口的确认信息丢失

✓两边都将阻塞（blocked）.

TCP 持续定时器

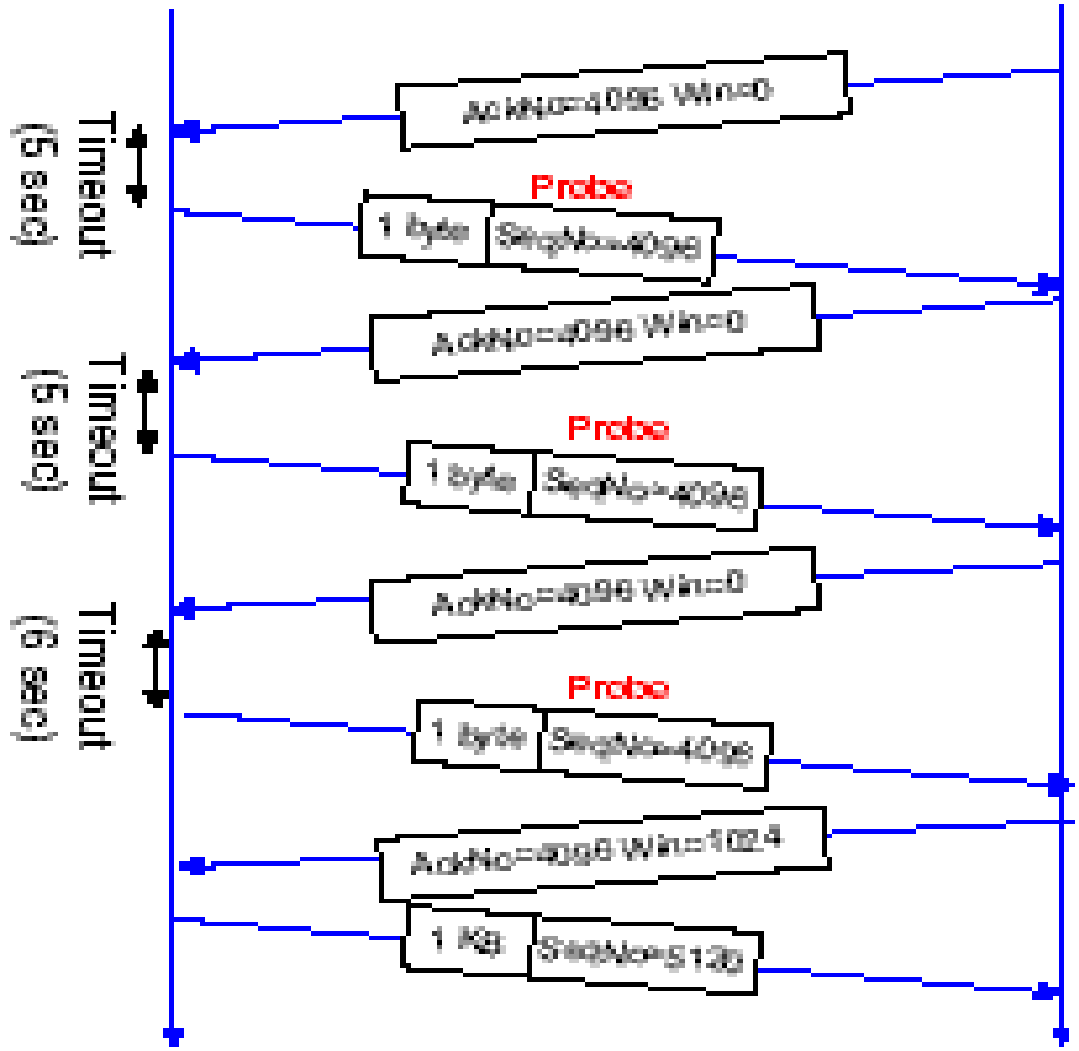
- 持续定时器在滑动窗口变为零时由发送方启动

持续定时器使用指数后退 (初始值为5秒)， 变化区间 **[5s**, 60s]。两次超时之间的时间间隔为:

5, 5, 6, 12, 24, 48, 60, 60, ...

窗口探测报文只有一个字节的数据 (窗口大小为零时**TCP**也要发送窗口探测报文)

TCP Persist Timer



TCP 存活定时器

- 当一个TCP连接闲置了一段时间后，存活定时器会自动提醒站点检查另一方是否还存在。
- 如果一个连接已经处于空闲状态2个小时，探测信息包将会被发送
- 假定一个探测包从A发到 B:

(1) **B** is up and running: **B** 以ACK应答

(2) **B** has crashed and is down: **A** 将每75秒发送一个探测包，共发10个. 如果**A** 仍然没有收到确认, 连接将会被关闭

(3) **B** has rebooted: **B** 会发送一个 RST 报文

(4) **B** is up, but unreachable: 参照 (2)

TCP性能优化

- 选择性确认（SACK）：指定目的站接收的分组块
- 随机早期丢弃 (RED) 策略：使丢包率更均匀，减少平均排队长度 和丢报率.
- 调度机制保护正常的信息流，拒绝不良的信息流.
- 显式拥塞通告 (ECN): 路由器明确通过拥塞指示位来通报拥塞，而不是通过丢包来判断.

Thanks !