计算机网络及应用

Computer Networks and Applications

第三章　传输层

面向连接的传输：TCP；拥塞控制原理；TCP拥塞控制

主讲：清华大学　贾庆山

教材：J.F. Kurose, K.W. Ross，Computer Networking: A Top-Down Approach, Addison Wiley, 7th Edition, 2017（机械工业出版社中文版，2018）

Special thanks to Prof. Kurose and Prof. Ross for presentation material

多选题　1分

可靠数据传输原理中用到了下面哪些技术

A　肯定/否定确认与重传

B　计时器

C　流水线

D　序号

E　校验和

TCP

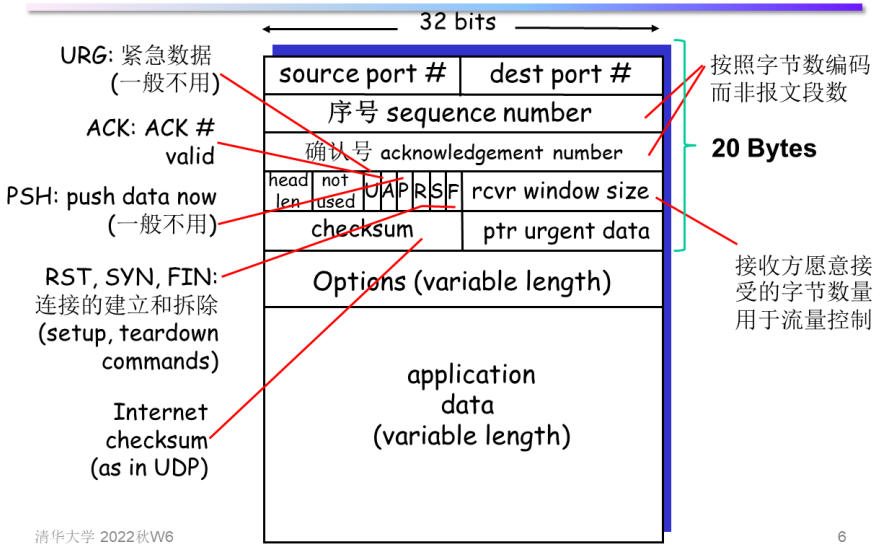
- 概述
- 数据段结构(TCP-PDU)
- TCP PDU中的SEQ和ACK
- TCP可靠数据传输的机制
- TCP流量控制
- TCP连接管理
  - 建立Establish
  - 拆除Teardown

TCP: 概述　RFCs: 793, 1122, 1323, 2018, 2581

- 成对出现
  - 一个发送方，一个接收方
- 可靠的保序的字节流
- 流水线
  - TCP 拥塞控制和流量控制设置窗口尺寸
- *发送和接收缓冲区*

- 全双工数据
  - 在同一个连接内双向数据流
  - MSS: 最大数据段尺寸
  - MSS<MTU, 1460/536/512B
- 面向连接
  - 三次握手(控制报文的交换)在数据交换前初始化发送方和接收方的状态
- 流量控制
  - 发送方不会淹没接收方
- 拥塞控制
  - 避免拥塞，从拥塞中恢复

TCP 报文段结构



TCP 序号和确认号

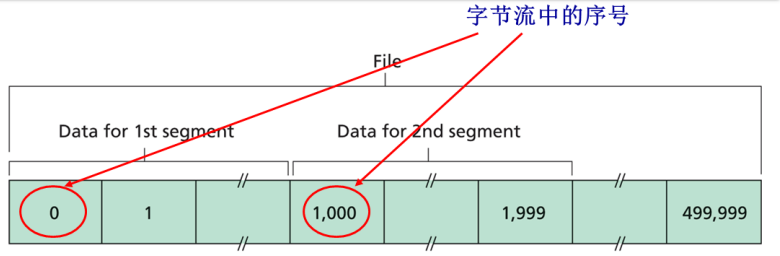


Figure 3.30 ♦ Dividing file data into TCP segments

TCP 序号和确认号

序号 Seq.#

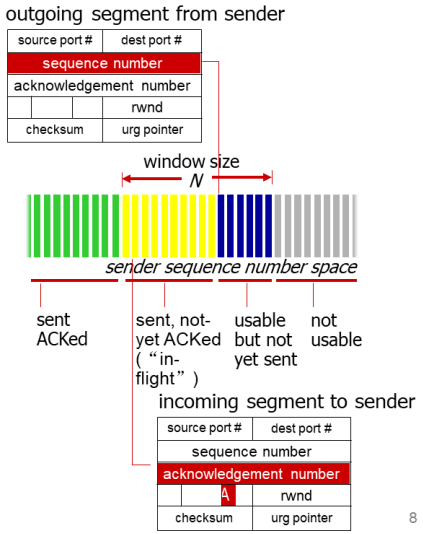
- 报文段数据首字节的字节流编号

确认号 ACK

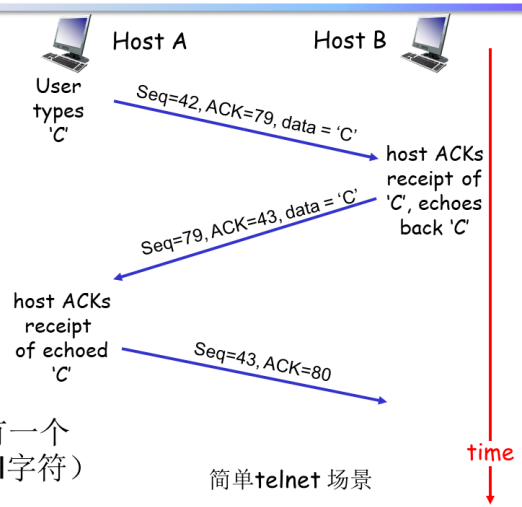
- 希望从对方收到的下一个字节的序号
- 提供累积确认 cumulative ACK

问题: 接受方如何处理失序报文段?

- TCP没有规定
- 由实现者自己决定
  - 缓存或丢弃



TCP 序号和确认号



数据段只有一个字节（ACII字符）

简单telnet 场景

如何选择合适的超时间隔

Timeout Interval > RTT

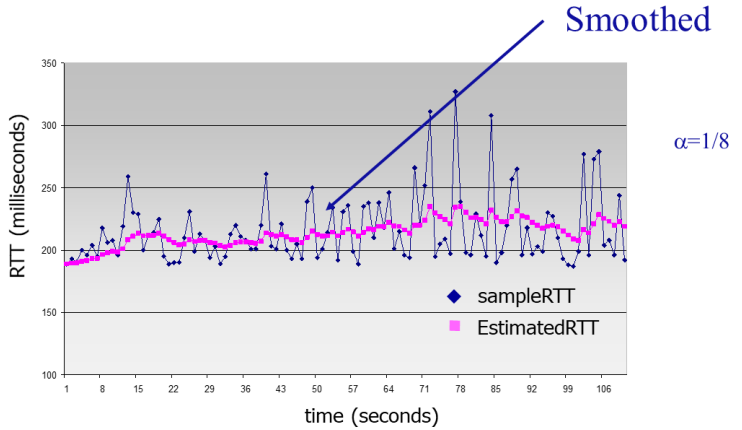
✓太短，过早重传
✓太长，延迟增加

问题：如何确定 RTT 的值？

- 动态更新
- 通过 EWMA (Exponential weighted moving average, 指数加权滑动平均，属于“时间序列回归分析”领域知识) 用新探测到的RTT来更新旧RTT

$$EstimatedRTT=(1-\alpha)\cdot EstimatedRTT+\alpha\cdot SampleRTT$$

随因特网的动态变化保持 RTT 随时更新



RTT between gaia.cs.umass.edu←→fantasia.eurecom.fr

更新超时间隔

RFC 2988

- 跟踪RTT的偏差量

$$DevRTT = (1-\beta)\cdot DevRTT+\beta\cdot|\ SampleRTT-EstimatedRTT\ |$$

e.g. $\beta = 1/4$

- 使超时间隔大于EstimatedRTT 是合理的

$$TimeoutInterval = EstimatedRTT+4\cdot DevRTT$$

EstimatedRTT加上足够大的安全裕量 → 超时间隔

TCP 可靠数据传输

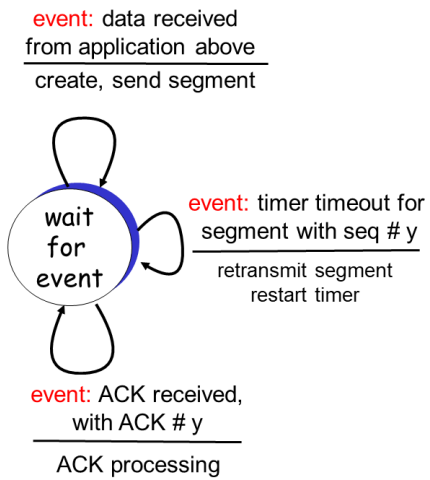
- TCP 在 IP 的不可靠的服务之上创建可靠数据传输服务
- 流水线方式传送报文段
- 累积确认
- TCP 使用单重传定时器
- 重传由下列事件触发
  - 超时事件 timeout events
  - 冗余确认 duplicated acks
    - 三次冗余确认，视为发生了丢包

TCP 可靠数据传输

先从简化的发送方开始，假定

- 单向数据传送
- 不考虑冗余确认
- 没有流量控制和拥塞控制

三种主要事件及响应动作



/*假设发送方不受TCP流量和拥塞控制的限制，来自上层的数据长度小于MSS，且数据传送只在一个方向进行*/

```
NextSeqNum=InitialSeqNumber
SendBase=InitialSeqNumber

loop (forever) {
   switch(event)

      event: data received from application above
         create TCP segment with sequence number NextSeqNum
         if (timer currently not running)
            start timer
         pass segment to IP
         NextSeqNum=NextSeqNum+length(data)
         break;

      event: timer timeout
         retransmit not-yet-acknowledged segment with
            smallest sequence number
         start timer
         break;

      event: ACK received, with ACK field value of y
         if (y > SendBase) {
            SendBase=y
            if (there are currently any not-yet-acknowledged segments)
               start timer
            }
            break;

} /* end of loop forever */
```

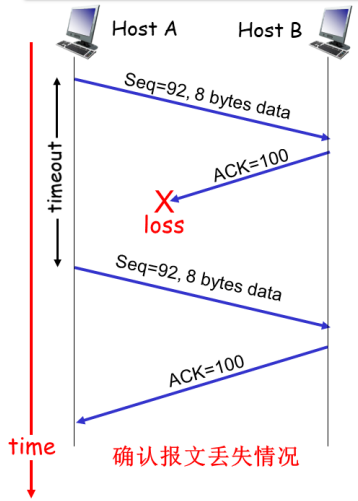
简化的 TCP 发送方

TCP实际过程参考实验三抓包分析

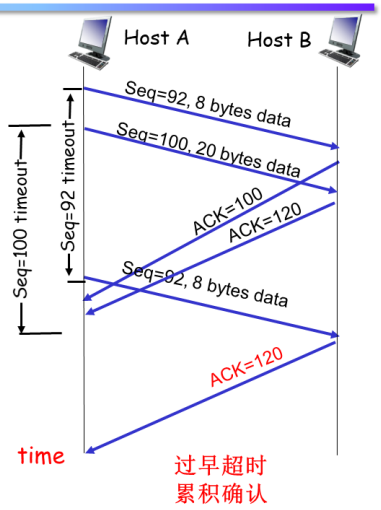
假设：对所有报文段有单一的定时器

Figure 3.33 ♦ Simplified TCP sender

重传机制中的一些场景

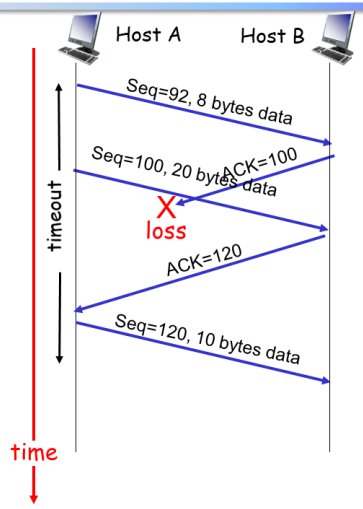


确认报文丢失情况



过早超时
累积确认

重传机制中的一些场景



累积确认避免了第一个报文段的重传

TCP确认(ACK)生成规则 [RFC 1122, RFC 2581]

event at receiver	*TCP receiver action*
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	*delayed ACK*. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending	immediately send single *cumulative ACK*. ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. # . Gap detected	immediately send *duplicate ACK*. indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate send ACK, provided that segment starts at lower end of gap

TCP的一些修改

1. 加倍超时间隔

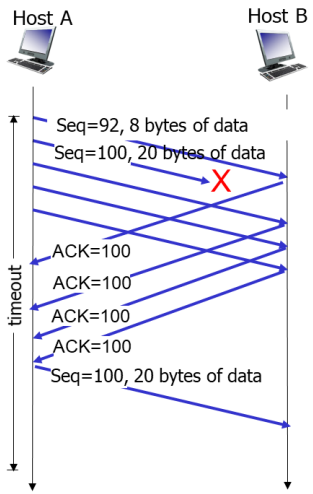
- 基本思想: 如果定时器超时，则增大定时器间隔为
  - TimeoutInterval = 2·TimeoutInterval
- 为什么? 考虑到拥塞发生
  - 类似以太网中的指数避让规则

2. 快速重传

- 基本思想: 三次冗余确认
  Duplicated ACKs 意味着有分组丢失，于是在定时器超时之前重传
- 为什么? 超时间隔可能太长
- 修改TCP简化模型中的Event 3

快速重传算法

```
event: ACK received, with ACK field value of y
  if (y > SendBase) {
    SendBase = y
    if (there are currently any not-yet-
       acknowledged segments)
       start timer
    }
  else{ /* a duplicated ACK for already
        ACKed segment*/
       increment # of duplicated ACKs
       received for y
       if (# of dupACKs for y==3){
            /*TCP fast retransmit*/
            resend segment with Seq# y
         }
       }
    break;
```

三次冗余确认后重传一个报文段

问: TCP是 GBN 还是 SR ?

GBN?

- Yes
  - 累积确认
- No
  - TCP 将失序的报文段缓存起来

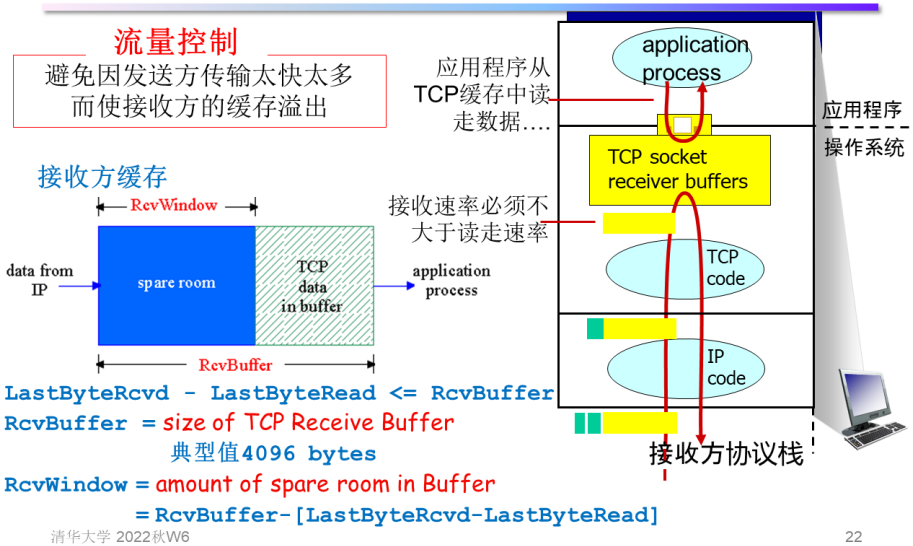
SR?

- Yes
  - 如果 ACK-k 丢失但ACK-k+m正确收到，不重传报文段（由累积确认机制保证）
  - 某些TCP修改版本采取选择确认机制
- No
  - 累积确认

因此 **TCP 组合了 GBN 和 SR**

TCP 流量控制

流量控制
避免因发送方传输太快太多而使接收方的缓存溢出



`LastByteRcvd - LastByteRead <= RcvBuffer`

`RcvBuffer` = size of TCP Receive Buffer
典型值4096 bytes

`RcvWindow` = amount of spare room in Buffer
= `RcvBuffer-[LastByteRcvd-LastByteRead]`

TCP 流量控制：工作原理

(假设 TCP 接收方丢弃失序的报文段)

接收方：明确地通知发送方（动态变化的）空闲缓存空间的数量
- TCP 报文段中的 **`RcvWindow`** 域

发送方：保持已发送的但未被确认的数据量在最近接收到的 **`RcvWindow`** 值以内
- 保证接收缓存不溢出

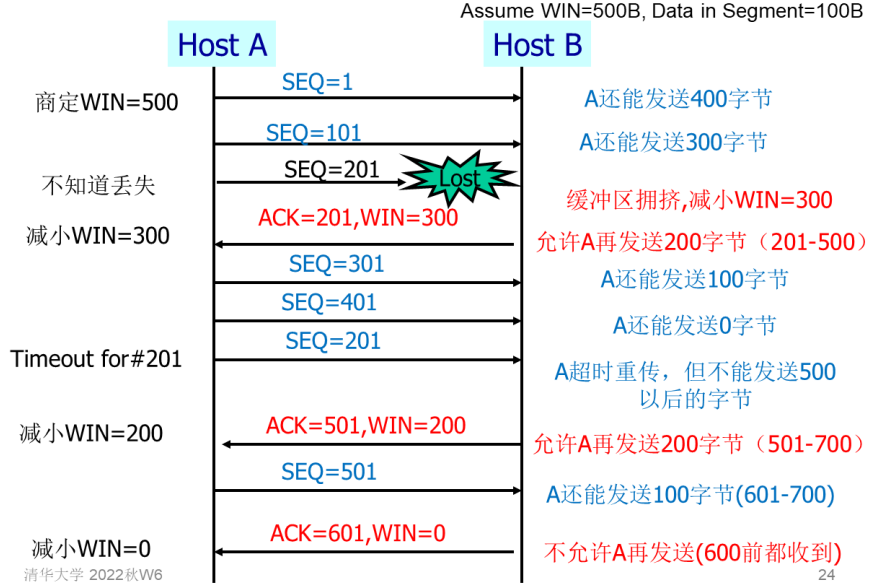
`LastByteSent-LastByteAcked<=RcvWindow`

UDP没有流量控制，尽最大速度发送，缓存溢出则丢弃

TCP 流量控制例子

右边WIN窗口值的确定由读数据速度及相关协议商定，并回送给左边

Assume WIN=500B, Data in Segment=100B



TCP 连接管理

Recall: TCP发送方、接收方在交换报文段之前要建立“连接”

- 初始化TCP变量
  - 序号
  - 缓存、流量控制信息(例如 **`RcvWindow`**)
- *client:* 连接的发起者

```
Socket clientSocket = new
Socket("hostname","port number");
```

- *server:* 由 client 联系

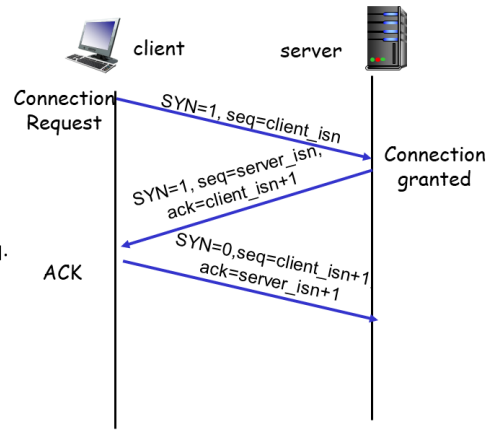
```
Socket connectionSocket = welcomeSocket.accept();
```

三次握手 Three way handshakes

Step 1: client 端系统发送 TCP SYN 控制报文段到 server
- 指定初始的 seq #

Step 2: server端系统收到 SYN, 以SYN ACK 控制报文回应
- ACKs received SYN
- 分配缓存、变量
- 指定 server-> receiver 初始 seq. #

Step 3: client 端系统发送 TCP SYN 控制报文到 server
- ACKs received SYNACK
- 分配缓存、变量
- 开始传送数据



TCP 连接管理 （续）

关闭一个连接

client closes socket:
`clientSocket.close();`

Step 1: client 端系统发送 TCP FIN 控制报文段到 server

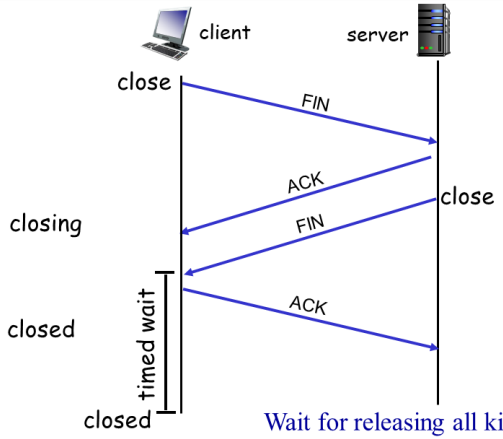
Step 2: server 收到 FIN，以 ACK 回应；关闭连接，发送 FIN。

Step 3: client 收到 FIN，以 ACK 回应
- 进入“计时等待”——将对收到的 FIN 以 ACK 回应

Step 4: server, 收到 ACK； 连接关闭。

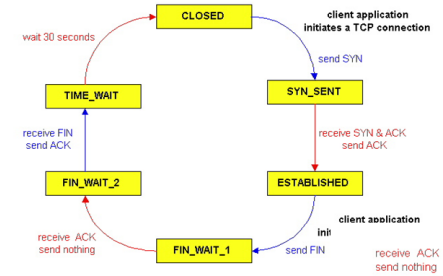
Note: 稍作修改，可以处理并发 FINs

TCP 连接管理 （续）

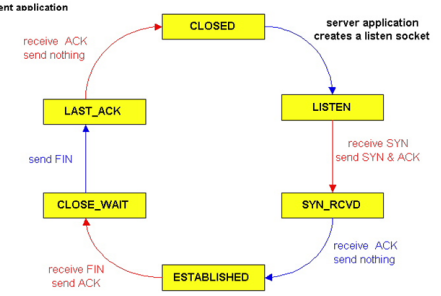


Wait for releasing all kinds of resources.
Typically, 30s, 1m, 2m.

TCP 连接管理 （续）



TCP client 生命周期



TCP server 生命周期

端口扫描器nmap与SYN洪泛攻击

- 向每个端口逐一发送SYN报文段
  - 返回SYNACK报文段：该端口处于“打开”
  - 返回RST报文段：该端口处于“关闭”
  - 什么都不返回：防火墙阻挡，不可达
- SYN洪泛攻击
  - 发送大量的SYN报文段，但不完成三次握手的第三步
  - 导致服务器不断为这些半开链接分配资源

提纲

- 传输层服务
- 复用和分解
- 无连接的传输: UDP
- 可靠数据传输原理
- 面向连接的传输: TCP
- **拥塞控制原理**
- TCP拥塞控制

拥塞控制原理

- 拥塞概述
- 拥塞的原因和代价
- 实现拥塞控制的方法

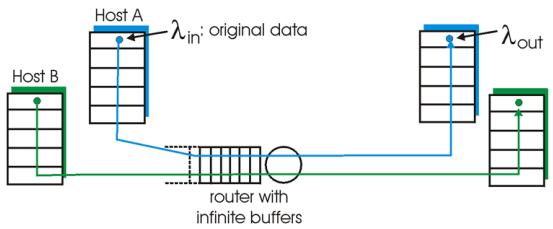
拥塞概述

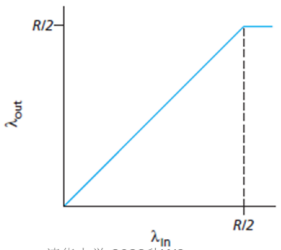
拥塞

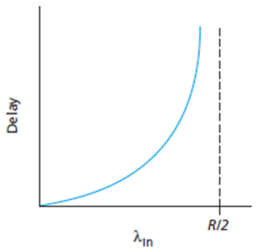
- 非正式定义: “too many sources sending too much data too fast for *network* to handle”
- 不同于流量控制 (flow control v.s. congestion control)
  - 流量控制导致的丢包发生在两端主机
  - 拥塞控制导致的丢包发生在核心设备路由器
- 征兆
  - 分组丢失 (路由器缓存溢出)
  - 长的时延 (在路由器缓存中的排队)
- Also a **top-10** problem in networking!

拥塞的原因/代价：场景 1

- 两个发送方,两个接收方
- 一个路由器, *无限大*缓存
- 无丢包误码，不重传





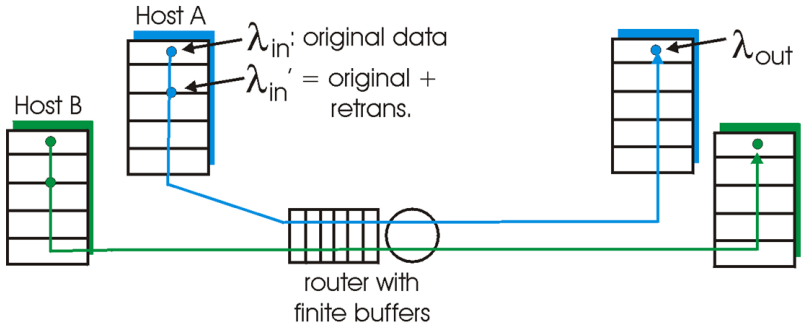


拥塞的代价

- 最大化可达到的吞吐量
- 但经历巨大的排队时延

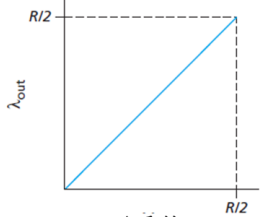
拥塞的原因/代价：场景 2

- 一个路由器, *有限的*缓存
- 发送方重传丢失的分组

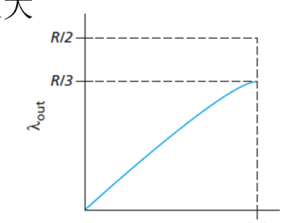


拥塞的原因/代价：场景 2

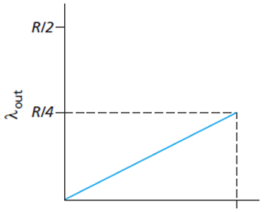
- 总是: $\lambda_{in} = \lambda_{out}$
- 仅当确认丢失时才重传 $\lambda'_{in} > \lambda_{out}$
- 对延误了的(并非丢失的)的分组进行重传使得λ'_{in}关于相同的λ_{out}变得更大



无重传



超时很大溢出重传



超时较小重复重传

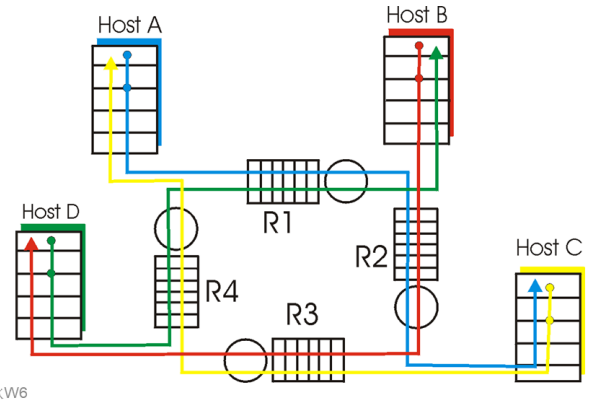
拥塞的代价:

- 更多的工作 (重传)
- 无用的重传: 链路上承载分组的多重拷贝

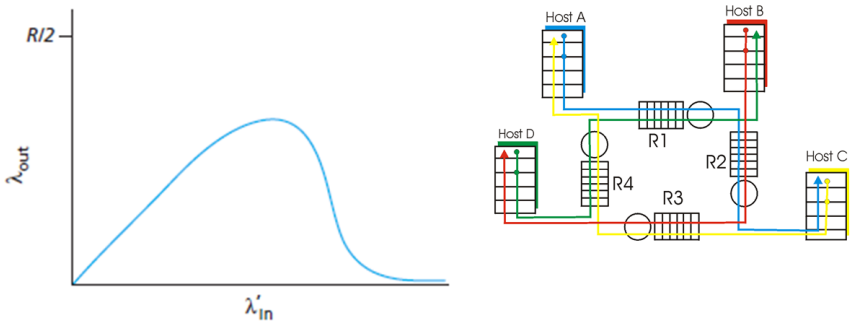
拥塞的原因/代价：场景 3

- 四个发送方
- 多跳路径
- 超时/ 重传

问: what happens as λ_{in} and λ'_{in} increase ?



拥塞的原因/代价：场景 3



另一种拥塞开销

- 当分组被丢弃后，任何用来传送该分组的上游传输容量都被浪费了!

实现拥塞控制的方法

两种主要方法来实现拥塞控制

端到端拥塞控制

- 没有来自网络的明确的反馈信息
- 拥塞由端系统从观测到的分组丢失与时延而推断出
- **TCP采用的方法**

网络辅助的拥塞控制

- 路由器向端系统提供反馈信息——更复杂的路由器!
  - 单比特指示拥塞状态（例如ATM）
  - 路由器把当前可行的数据率明确反馈给发送方

提纲

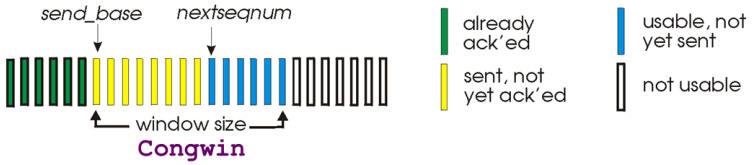
- 传输层服务
- 复用和分解
- 无连接的传输: UDP
- 可靠数据传输原理
- 面向连接的传输: TCP
- 拥塞控制原理
- **TCP拥塞控制**

TCP 拥塞控制

- 问题描述
- 原理
- 慢启动机制

TCP 拥塞控制的问题描述

- 端到端控制 (没有网络层辅助)
- 传输速率由拥塞窗口的大小 **Congwin** 所限制



LastByteSent-LastByteAcked<=min{Congwin,RcvWin}

- w 个报文段, 每个具有MSS 字节，在一个RTT中发送

$$\text{throughput} = \frac{w * MSS}{RTT} \text{ Bytes/sec}$$

假设传输延迟相对于RTT可忽略

问题：怎么设置合适的**Congwin窗口**大小，使其既充分利用带宽，又避免拥塞？