传输层

笔记源文件：Markdown，长图，PDF，HTML

1. 概述

1️⃣传输层功能的：

1. 实现进程间的逻辑通信(相比下，网络层实现的是主机之间的逻辑通信)
2. 差错检测
3. 多路复用/分解
4. 提供无连接/面向连接的服务

2️⃣传输层协议：TCP和UDP，在端系统(而非路由器)中实现

3️⃣传输层基本工作过程：

1. 发送端应用层：应用进程将报文丢给传输层
2. 发送端传输层：将应用层来的报文转化为传输层报文(切块+加上传输层首部)，完成后丢给网络层
3. 发送端网络层：封装成分组，发出去
4. 接收端网络层：接收，并提取出传输层报文段，送给传输层
5. 接收端传输层：处理收到报文，使得报文中的数据可供应用进程使用
6. 接收端应用层：收到并使用数据

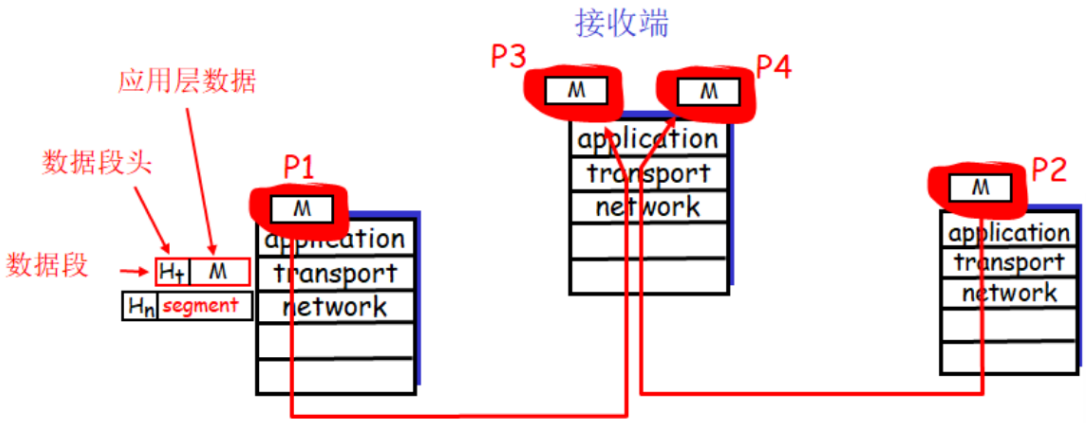
4️⃣IP/TCP/UDP概述

1. IP概述：每个主机至少一个IP，IP协议是尽力而为的不可靠协议，不保证数据交付的顺序和完整性
2. TCP/UDP职责：
   - 多路复用/分解：IP交付/主机交付$\xrightarrow{\text{扩展}}$进程间的交付服务
   - 提供完整性检查：根据报文首部的差错检查字段
3. UDP：也是不可靠的，只能提供差错检测+进程端端交付

2. 多路复用/分解

2.1. 基本概念

1️⃣套接字：网络和进程间数据传递的门户，由进程持有，传输层$\xleftarrow{\text{收到数据}}\xrightarrow{}$套接字$\xleftarrow{\text{收到数据}}\xrightarrow{}$进程



2 多路分解(接收端)：传输层检查报文某些字段→识别出接收套字→将报文交付给正确套接字

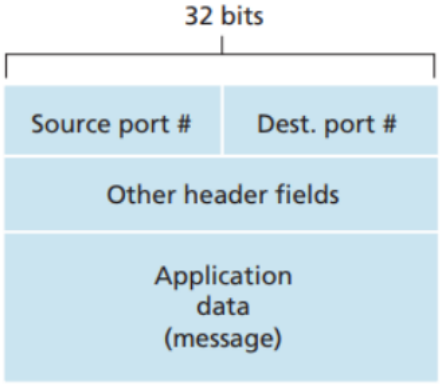
3 多路复用(发送端)：从套接字处收集数据块→为每个数据块封装上首部(用于分解)→通过网络层发送

就好比课代表发作业(多路分解)，收作业给老师(发送端)

2.2. 传输层能多路分解/复用的要求

1 套接字有唯一标识符

2 每段有特殊字段(源端口号+目的端口号)，标识索要交付到的套接字



➕端口号：16位，0-1023的为周知端口号，大于1023的是用户端口号

2.3. 两种多路复用/分解

	无连接的	有连接的
协议	UDP	TCP
套接字格式	`<目的IP><目的端口号>`	`<源IP><目的IP><源端口号><目的端口号>`

1 只有套接字两项/四项完全相同时，报文才会被送到相同套接字

2 无连接的应用程序，自动分配端口号

3. 用户数据报协议(UDP)

3.1. 概述

1 数据报：UDP的段

2️⃣UDP完成工作：有且仅有多路复用/分解+差错检测，也就是传输层最基本的工作，直接和IP交互

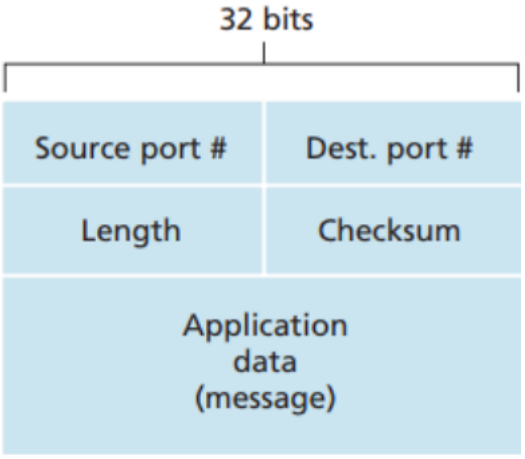
3️⃣UDP的特点

1. 简单无连接：传输数据前双方不事先连接
2. 是best effort服务：尽最大努力传输，但不保证可靠性/顺序
3. 速度快
4. 首部小：只有8字节，开销小

4️⃣UDP的应用：NFS，流式多媒体，DNS

3.2. UDP报文段结构

段头+数据，其中段头只有32bit，含源端口号+目的端口号+长度+校验



3.3. UDP校验&checksum

1️⃣目的：检测UDP报文从源到目的过程中，是否发生改变

2️⃣特点：检错能力很弱

3️⃣检错步骤：将段的内容看作16为二进制数集合

- 发送端：
  1. 获得校验和：所有16位二进制数相加→截取低16位→结果按位取反
  2. 将校验和输入UDP校验和字段Checksum
- 接收端
  1. 对段内容所有16位二进制相加，截取低16位
  2. 将结果+Checksum，如果全为1111111111111111就没出错

4. 传输控制协议TCP

观前提醒：ACK是指确认收到数据的信号

4.1. TCP概述

1️⃣特点

1. 点对点：发送/接收方都只能有一个

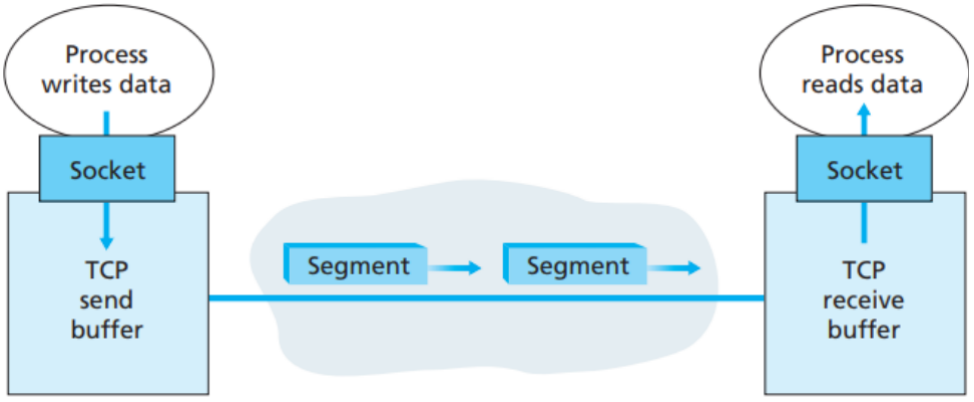
2. 可靠的

3. 无报文边界：字节以有序流发送数据，而不是分成独立报文

4. 流水线式：通过设置窗口大小，实现拥塞/流量控制

5. 全双工：数据可以同时双向传输，UDP同样

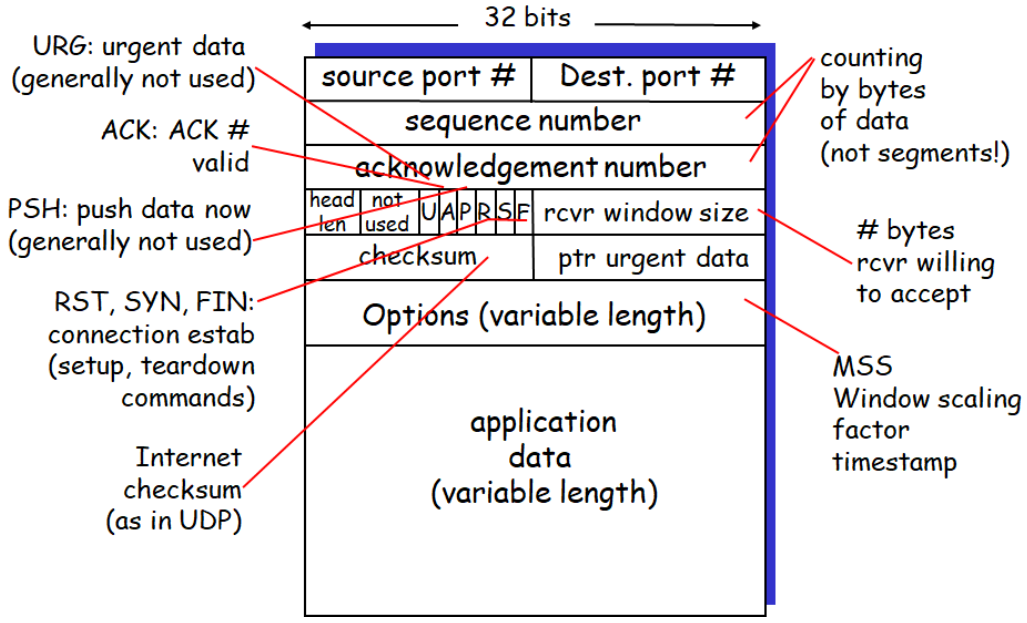
2️⃣结构概览



3️⃣功能：三控一管，可靠性/流量/拥塞控制，连接管理

4.2. TCP段文

4.2.1. 段文格式：一共五行20字节



1️⃣第一行：源端+目的端口号

2️⃣第二行：序列号，TCP的传输是一个个字节流送的，要给每个字节编号保证按序交付

3️⃣第三行：确认号，TCP有确认机制(接收端$\xrightarrow{ACK}$发送端，当ACK=N则N-1及以前的数据都已收到

4️⃣第四行：

1. 首部长度
2. 保留字段：占6位，忽略不计
3. 标志位

标志位	标志位=0	标志位=1
U(URG紧急)	紧急指针字段无效	有效，报文中有紧急数据，优先级高
A(ACK确认)	确认号字段无效	有效，TCP连接建立后，所有ACK=1
P(PSH推送)	\	收到PSH=1的报文，会优先上交给应用进程
R(RST复位)	\	当RST=1时，说明TCP连接崩溃，需要释放连接重传
S(SYN同步)	\	当SYN=1时，表示整个报文是一个连接请求/连接接收报文
F(FIN终止)	\	当FIN=1时，表示数据传输结束，就地释放连接

4. 窗口字段：**明确指定了目前允许对方发送的数据量**

5️⃣ 第五行

1. 校验和字段：检测范围包括首部+数据，计算校验和时需要在TCP报头前加上12B伪首部
2. 紧急指针字段：前面已说，由URG控制

6️⃣ 第六行开始：选项字段，长度可变，内容可选(最早的内容为MSS)

7️⃣ 最尾部：填充字段，填充使整个首部长度为32bit整数倍

4.2.2. 其他

1️⃣ 最大段大小(MSS)：536字节

2️⃣ TCP段文 <--协同工作--> IP数据报

1. TCP报文首部不包含IP地址，IP地址在IP数据报首部
2. IP层负责处理目的IP：TCP协议在发送数据时，依赖于IP协议来确定数据报的目的地

4.3. TCP的可靠性控制：丢包重传

4.3.1. 发送窗口(发送缓冲区)

1️⃣ 数据结构：开始指针send_base + 窗口大小n + 下一个序列号nextseqnum

2️⃣ 发送窗口的行为

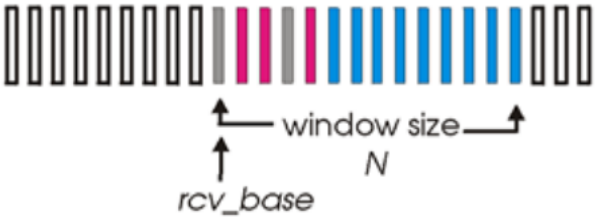
1. 应用层请求通过传输层发送数据，先检查nextseqnum是否有效
2. 有效的话，就将nextseqnum封装成TCP段发送，同时nextseqnum++

3. 一直移动到nextseqnum - send_base = N(窗口满)，无法继续发应用层塞的数据了
4. 直到接收方发回ACK，根据ACK，send_base右移，跳过已确认收到的段

3️⃣ 丢失段和发送窗口

1. 发送窗口一定包含丢失段
2. 如何判定窗口中的段已丢失：发回的ACK包含了段序号，超时后ACK还没某段序号就重传

4.3.2. 接收窗口(接收缓冲区)



1️⃣ 数据结构：开始指针rcv_base + 窗口大小n，其中rcv_base指向

2️⃣ 核心机制：**通过发送期待接收下一个段的序列号，以此来确认收到当前段**

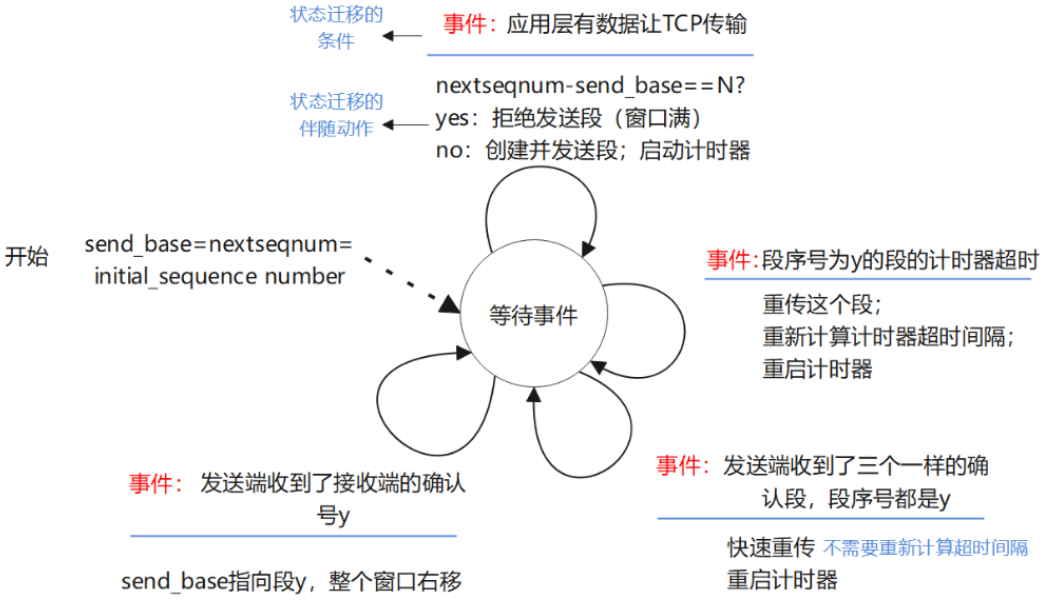
3️⃣ 接收窗口的行为

1. rcv_base指向接收方期待收到的段的段号(第一个灰色)
2. 串口内不断收到段(红色)，在期待段到达前，先将红色的缓存起来(蓝色的为空闲接收缓存)
3. 期待段到后，连通所有红段都一起上交应用层
4. rcv_base来到下一个期待收到的段(第二个灰色处)

4.3.3. 发送端的可靠性控制

1️⃣ 控制机制：

1. 只有一个状态即等事件，状态迁移结尾等事件→等事件
2. 假设无流量/拥塞控制，应用层给到的数据长度小于MSS，数据传输单向



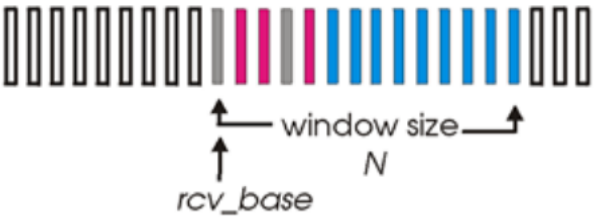
2️⃣ 快速重传：代表了轻度拥塞(超时对应重度拥塞)

1. 接收端没收到期待包，不论接下来收到了什么都只会返回期待包的ACK，因为TCP要求数据有序
2. 当返回相同ACK三次，就可认为这个ACK所代表的包已经丢了
3. 启动快速重传

3️⃣ 伪代码

```
send_base = init_sequence number
nextseqnum = init_sequence number
loop(永远){
    switch(事件)
    事件:应用层有数据让TCP传输
        if(nextseqnum-send_base<N){
            创建段序号为nextseqnum的段
            启动计时器
            将段发给IP层
            nextseqnum = nextseqnum +数据长度/*段序号是跳跃式的*/
        }else{
            拒绝发送段
        }
    事件:段序号为y的段的计时器超时
          重传这个段y
        重新计算计时器超时间隔
        重启计时器
    事件:接收到ACK，字段值为y
          if(y>send_base){/*段在发送窗口内*/
              取消掉段y之前所有的段的计时器
              send_base = y/*窗口右移*/
          }else{/*这里指的是y=send_base，接收端还没有收到y*/
              对ACK字段为y的计数器+1
              if(计数器的值==3){
                   快速重传段y
                   重启段y的计时器
              }
          }
}
```

4.3.4. 接收端的可靠性控制



收到段的特征	TCP接收端动作
有序到达 无间隙 其他段都已确认	延迟等待下一个段0.5s 1. 期间如果下一段来了则二者一起确认 2. 没来的话就发送ACK

收到段的特征	TCP接收端动作
有序到达 无间隙 有一个ACK在做延时	就是上述“下一段”，二者立刻一起确认ACK
乱序到达 有间隙 (如上图红色部分)	立即发ACK，内容为期待段的段号
乱序到达 但填满了某些间隙	目的在于补齐gap，间隙成因有两种(期待段空缺+其他乱序段间空缺) 1. 收到期待段(左灰)：连同右边连着的红段送给应用层，窗口右移 2. 收到其他段(右灰)：收到段变红，返回ACK(内容为期待段段号)
收到段位于窗口左侧	丢弃段

4.3.5. TCP往返时间和超时

1️⃣ 一些概念

1. 超时时间间隔：数据发送后，在这个时间内还未收到ACK，就要重传
2. RTT：数据包的往返时间，理想超时时间间隔应该略大于RTT
3. 安全边际：略大于RTT，略大于的这部分就是安全边际

2️⃣ 如何预测RTT：一般$\alpha = 1/8, \beta = 1/4$

1. $\text{sample}RTT$：策略从传输→收到ACK的时间差，缺点是波动大
2. $\text{estimate}RTT$：用EWMA平滑估计RTT

   $$\text{estimate}RTT_n = (1-\alpha)\text{estimate}RTT_{n-1} + \alpha\text{sample}RTT$$

3. deviation：估计前两者的差距，来反映波动性大小

   $$\text{deviation}_n = (1-\beta)\text{deviation}_{n-1} + \beta|\text{sample}RTT - \text{estimate}RTT|$$

3️⃣ 超时时间间隔$= \text{estimate}RTT + 4 * \text{deviation}$

4.4. TCP流量控制

4.4.1. 概述

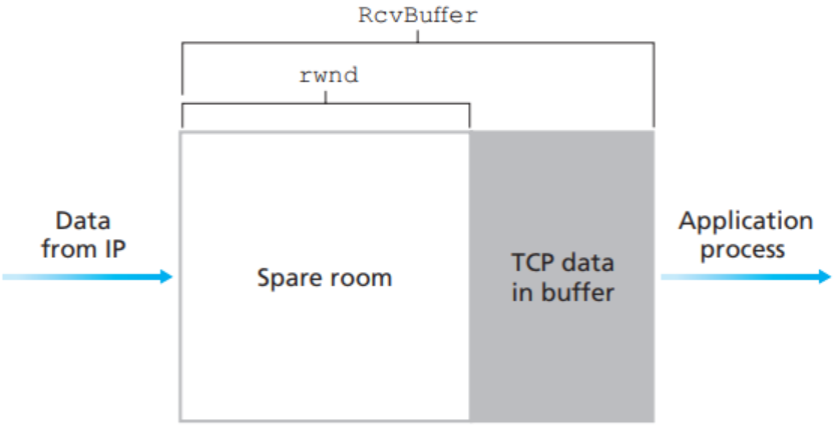
1️⃣ 背景：发送太快，接收端应用程序读取太慢，就会造成接收窗口溢出

2️⃣ 流量控制的含义：让发送方发送速率=接收方应用程序读取速率，防止溢出

3️⃣ 核心机制：

1. TCP段头中有一个字段表示接收窗口大小
2. 接收窗口会给发送方指明，接收方还有多少可用缓存

4.4.2. 控制过程



1️⃣接收端行为：A通过TCP连接向B发文件，B为该连接分配接收缓存，B的应用不断从缓存取走数据

2️⃣接收有关变量

1. `LBRead`：缓存中被读走的最后一个段的段号
2. `LBRcvd`：缓存中刚收到的段的段号
3. `RcvBuffer`：缓存大小，满足`LBRcvd-LBRead ≤ RcvBuffer`
4. `rwnd`：接收窗口(空闲缓存)，等于`RcvBuffer-[LBRcvd-LBRead]`，初始为`rwnd=RcvBuffer`

⚠️主机A需要明白B的`rwnd`还有多大，通过将`rwnd`放到B传回给A的报文的接收窗口字段即可

3️⃣发送端有关变量

1. `LBSent`：最后一个被送出的字节
2. `LBAck`：最后一个被确认的字节，`LBAck-LBSent`就是发出但未收到确认的数据

4️⃣流量控制的核心：在主机A的整个生命周期，保证LBAck-LBSent ≤ rwnd

4.4.3. 零窗口探测

1️⃣Bug所在：

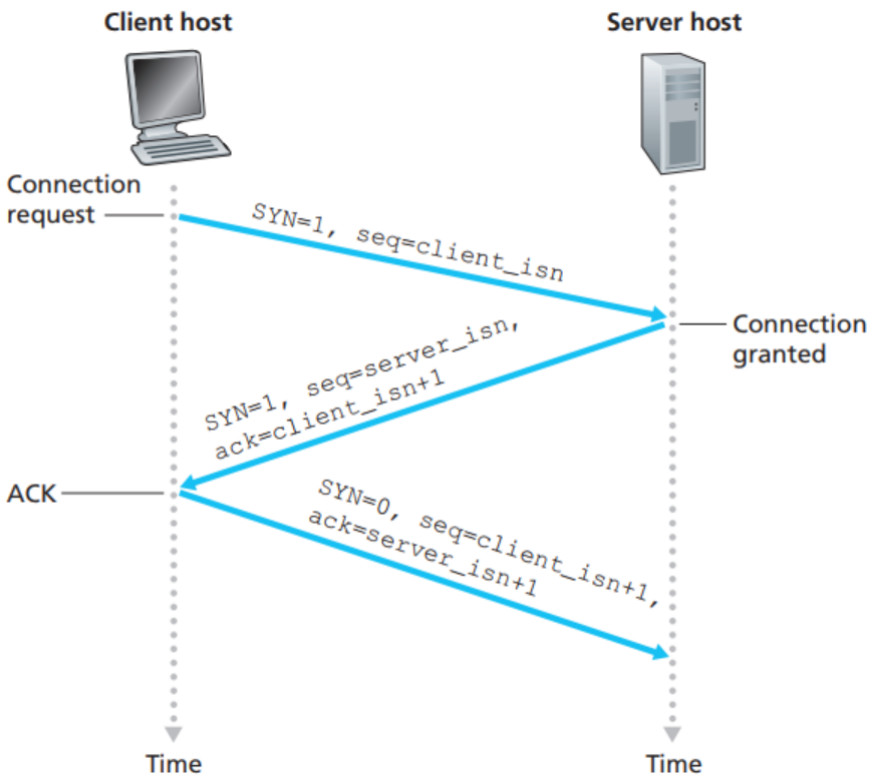
1. 当B端`rwnd=0`时，发送端A必定会停止发送
2. B然后不会向A反馈`rwnd=0`的变化，因为TCP只在发信/ACK时才发送报文
3. 之后B的应用程序会取走缓存使得`rwnd>0`，但是A就也无法得知了

2️⃣Debug所在

1. 不论`rwnd=0`与否，A都定期发送只包含1字节的探测报文段
2. 接收了探测报文的B端，无论如何都不可能是零窗口了，即可恢复数据传输

4.5. TCP的连接管理

4.5.1. 开启连接：三次握手

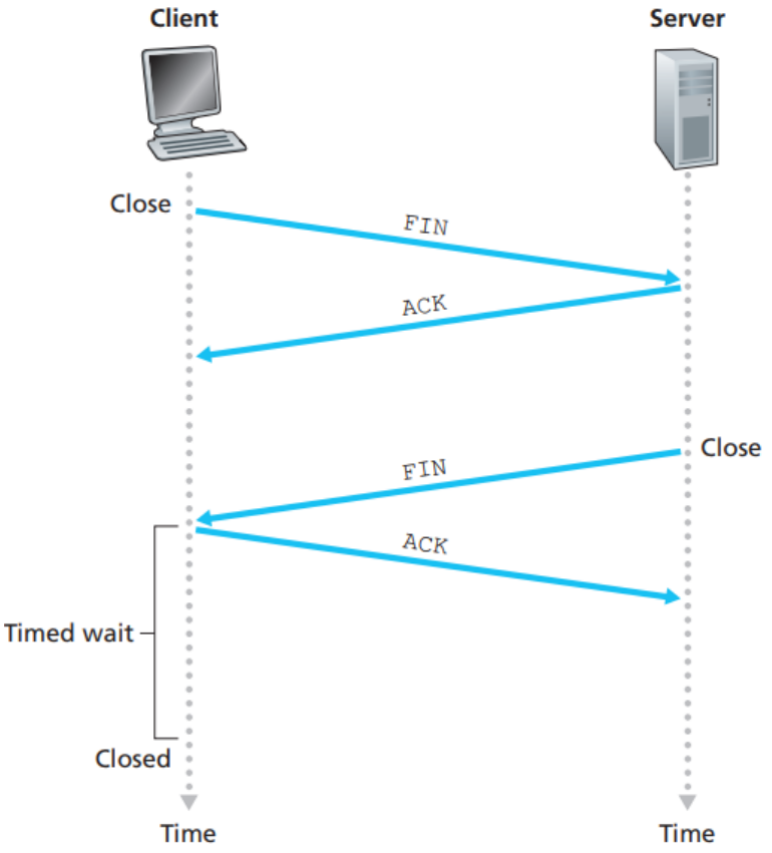


(Client=发送端，Serve=接收端)

握手	方向	报文类型	Client初始序列号	Server初始序列号	确认号
第一次	C→S	SYN	`J`	\	\
第二次	S→C	SYN+ACK	\	`K`	`J+1`
第三次	C→S	ACK	\	\	`K+1`

1. 三次握手后TCP连接建立，Clinet开辟缓存，开始传输
2. 补充说明：`J`和`K`是随机生成，ACK/SYN/FIN报文的报头对应标志位为1
3. DDos攻击：永远吊在第三次握手未完成状态。服务器不断开辟内存，直到服务器崩溃

4.5.2. 关闭连接：四次握手

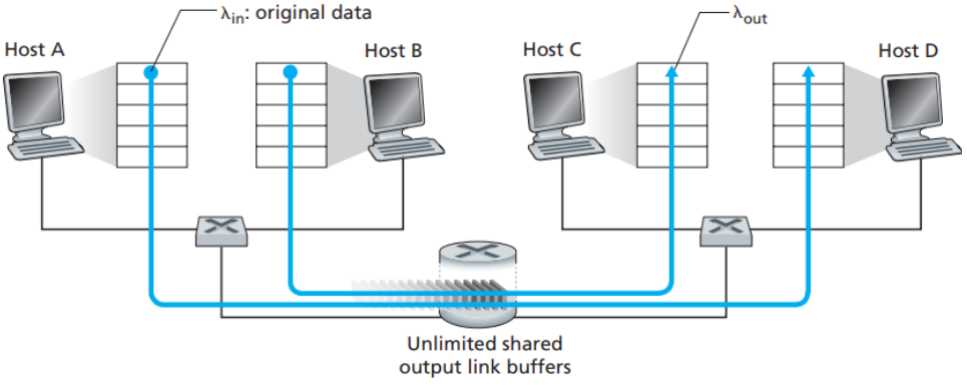


握手	方向	报文类型	C-S行为
第一次	C→S	FIN	所以爱会消失的，结束吧
第二次	S→C	ACK	好的
第三次	S→C	FIN	那就结束吧
第四次	C→S	ACK	好的，达成共识

四次握手后，要等一段时间TCP连接才正真关闭

4.6. 拥塞控制

4.6.1. 拥塞控制概述



1. 拥塞：是指路由器的拥塞
2. 表现：丢包(路由器缓冲区溢出)，长延迟(一直在路由器缓冲区中排队)

3️⃣ 成因：

1. 长延时：发送端发送总速度>路由器交换能力，**延时太大了会被误以为丢包，而进行无意义重传**
2. 丢包：发送端塞给路由器的数据>路由器有限的缓存

4️⃣ 分类

1. 网络帮助的拥塞控制：交换机检测到拥塞后直接控制，可为交换机→发端，交换机→收端→发端
2. 端到端的拥塞控制：端系统功能强(路由器相对弱)，端根据网络反馈调节拥塞(超时/快速重传)

4.6.2. ATM(异步传输模式)的拥塞控制

Old Fashined

1️⃣ ATM业务类型

类型	名称	描述	特点
ABR	Available Bit Rate	有效位率服务，用于视频	可能丢包，保证最小带宽
CBR	Constant Bit Rate	用于实时语音通信	不丢包，不拥塞控制
VBR	Variable Bit Rate	变动位率服务	不丢包，不拥塞控制
UBR	Unspecified Bit Rate	有资源则使用，无资源则丢包	免费使用，无拥塞控制

2️⃣ 信元：ATM的数据单元

1. 数据信元
2. 资源管理信元：存放拥塞信息，同工厂几十个信元里就有一个资源管理信元

3️⃣ ATM的ABR拥塞控制方法

1. 信元头部加CI(拥塞指示)和NI位(不增加速率)：拥塞后CI=1发端会降低发送速率，NI用于让发端速率不再增加
2. ER设置：这是在资源管理信元的字段，告诉发端可以按多大速率发数据，有多个则取最小
3. EFCI：位于数据信元，检测到拥塞后置1

4.6.3. TCP的拥塞控制

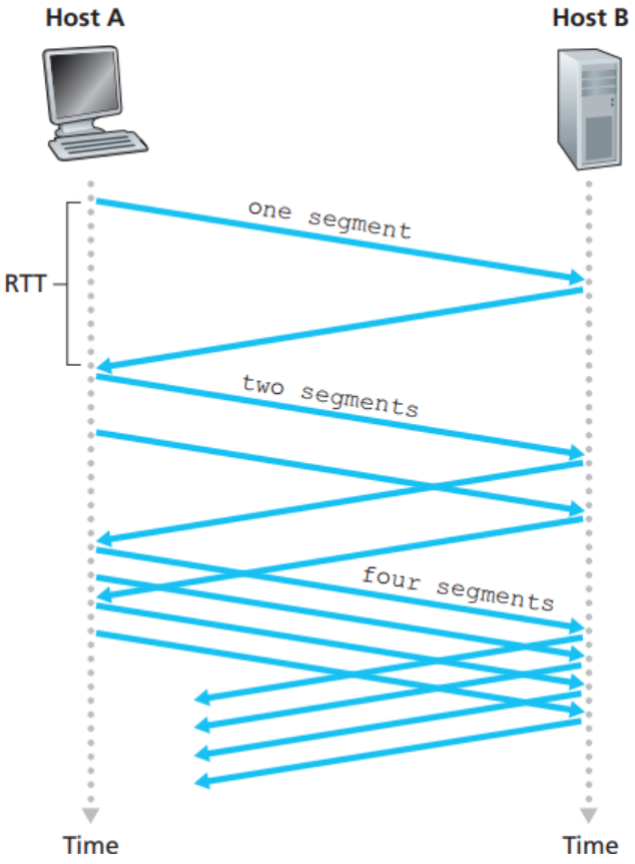
1️⃣ 两种判断：超时(重度拥塞)，收到三个相同ACK(轻度拥塞)

2️⃣ 探测拥塞：慢启动(不断*2)+拥塞避免(改为+1)

```
第1个RTT：发1个探测段，收到1个ACK则没拥塞
第2个RTT：发2个探测段，收到2个ACK则没拥塞
第3个RTT：发4个探测段，收到4个ACK则没拥塞
.........n次试探后还没拥塞.......
第n+1个RTT：发$2^n$+1个测试段
```

3️⃣ TCP慢启动

1. 慢启动过程



2. 慢启动伪代码

```
threshold=适当的值(10、20...不要太大) //阈值，区分慢启动和拥塞避免
Congwin=1 //拥塞控制时，使用的窗口大小
for(每个确认段) //每收到一个ACK窗口就+1，第一轮1个ACK，第二轮2个，第三轮4个...
   Congwin++
until(丢包&&ngWin>=threshold)
```

4️⃣ 拥塞避免

1. Tahoe拥塞避免算法伪代码：不合理的点在于没区分轻度/重度拥塞，一股脑将Congwin=1

```
//慢启动结束
while (没有丢包) {
  每w个段被确认:
       Congwin++//每个RTT，窗口+1线性增加
}
//如果丢包则退出循环
threshold = Congwin/2
Congwin = 1
//重启慢启动
```

2. Reno拥塞算法伪代码：吞吐率更高，震荡更小

```
//慢启动结束
while (没有丢包) {
```

```
  每w个段被确认：
      Congwin++//每个RTT，窗口+1线性增加
}
//如果丢包则退出循环
threshold = Congwin/2
if(因为超时丢包){//重度拥塞
    Congwin = 1
   重启慢启动
}
if(因为收到三个相同确认段丢包){//轻度拥塞，只需要快速恢复
    Congwin = Congwin/2
    goto: while循环
}
```

5️⃣ 平均吞吐率≈ $\frac{1.22MSS}{RTT\sqrt{L}}$，$L$为丢包率，$MSS$为最大段大小

6️⃣ 算法特点：

1. 线性增加窗口，丢包后指数减少窗口
2. 有效，收敛，公正，友好