TCP数据传输、连接管理和计时器

教材目录

3.3 数据链路协议：可靠数据传输

- 停等协议
- 滑动窗口协议：GBN和选择重传

6.2.1 TCP概述

6.2.2 TCP段格式

6.2.4 TCP可靠传输

6.2.5 TCP流量控制

TCP可靠数据传输

6.2.3 TCP连接管理

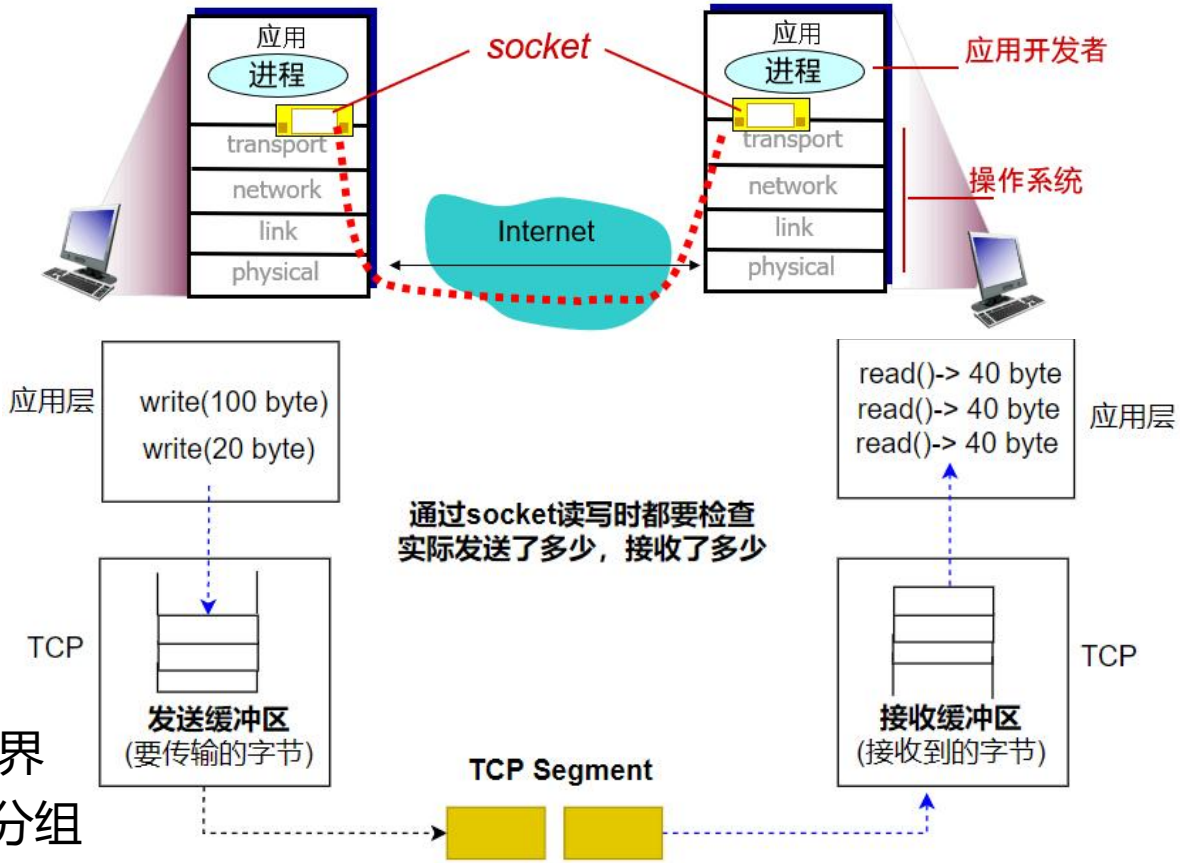
6.2.7 TCP计时器

6.2.6 拥塞控制

principle of robustness: be conservative in what you do, be liberal in what you accept from others.

TCP概述(RFC 793定义，最新的标准为2022年8月发布的RFC9293)

- Internet的运输层提供两种通用的服务
  - UDP(User Datagram Protocol)提供不可靠的数据传输
  - TCP(Transport Control Protocol)在RFC 793定义，在IP提供的无连接方式的尽力递交的数据传输服务上为应用层提供**面向连接的、 端到端(点对点)的全双工的字节流方式的可靠**数据传输服务
- 面向连接的：
  - 首先建立一条连接，然后数据传输，最后关闭连接
  - 不支持组播、广播，支持端到端的单播传输
- 可靠数据传输：采用序号、确认和重传机制
- 流量控制：防止发送过快以至于另一方缓冲区用完的情况
- 拥塞控制：防止以超过网络处理能力的速度发送
- 全双工(Full-Duplex)：
  - 捎带确认
  - 可以关闭一个方向的**数据流**(Half-Duplex)
- TCP的数据传输服务是**面向字节流**，意味着用户数据没有边界
  - 应用程序只看到一个一个的字节，而不是一个一个的分组
  - TCP的发送方和接收方都采用缓冲区以便积累足够多的数据发送以及缓存收到的数据



TCP段(segment)格式

TCP段： 20字节的固定头部+最多40字节可选头部+用户数据

- 数据偏移(头部长度) : 4比特
  - 以32比特为单位，限制最长 4*15 = 60字节
  - 选项：可选，最多40字节，提供相应扩展机制
  - 填充：保证用户数据以32比特边界开始
- 用户数据长度= IP头部的总长度(16bit)- IP头部长度-TCP头部长度,最大值 = 65535-20-20=65495
- **MTU(Maximum Transfer Unit)**：**链路层帧携带的用户数据部分**的最大长度。以太网的MTU = 1500 字节
- **最大分段大小MSS**(Maximum Segment Size): **TCP段中用户数据**的最大长度，以太网上的MSS <= 1500 - 20 -20 = 1460字节

IP头部： **源IP地址 + 目的IP地址 + 协议号** + **总长度 + 头部长度** + IP头部检验和

32 位

源端口号										目的端口号
顺序号										
确认号(ACK置位时有效)										
数据偏移	保留(4b)	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	窗口大小
校验和										紧急指针(如果URG=1)
选项（0个或多个字节）										填充
用户数据 （可选）										

TCP段(segment)格式

TCP段： 20字节的头部+最多40字节可选部分+用户数据

- **源端口号+目的端口号**与IP头部的**源IP和目的IP地址以及协议(TCP or UDP等)**标识一条TCP连接
- 标志位

SYN：同步顺序号，建立TCP连接

FIN：无数据发送，连接释放

RST：对TCP连接进行复位

PSH：一般当除了正在发送的TCP段外，目前发送缓冲区不再有数据等待发送时设置，要求接收者马上递交

ACK：确认字段有效标志

URG：紧急指针有效标志

ECE(ECN-Echo): 通知拥塞出现

CWR(Congestion Window Reduced) 发送者已减少拥塞窗口

- 检验和(checksum):TCP段是否有差错

IP头部： **源IP地址 + 目的IP地址 + 协议号** + 总长度 + 头部长度 + IP头部检验和

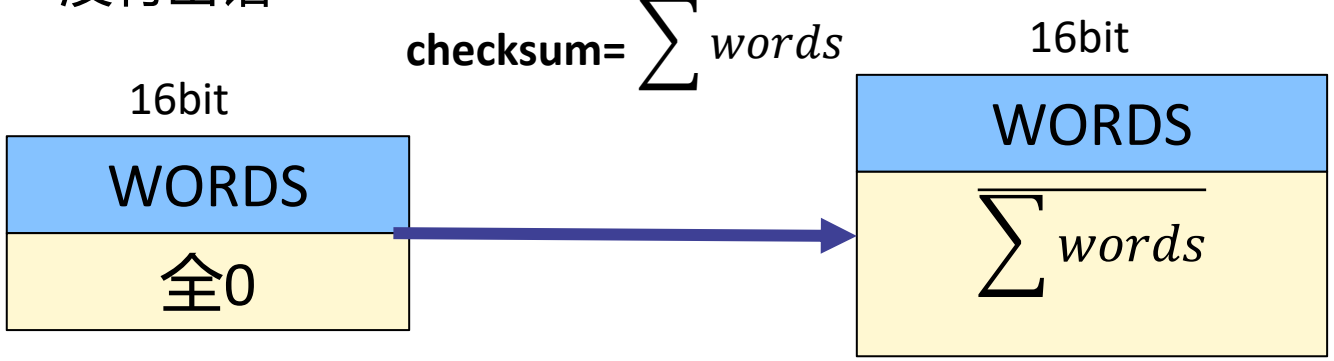
32 位

源端口号										目的端口号
顺序号										
确认号(ACK置位时有效)										
数据偏移	保留(4b)	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	窗口大小
检验和										紧急指针(如果URG=1)
选项（0个或多个字节）										填充
用户数据 （可选）										

TCP段(segment)格式：检验和算法

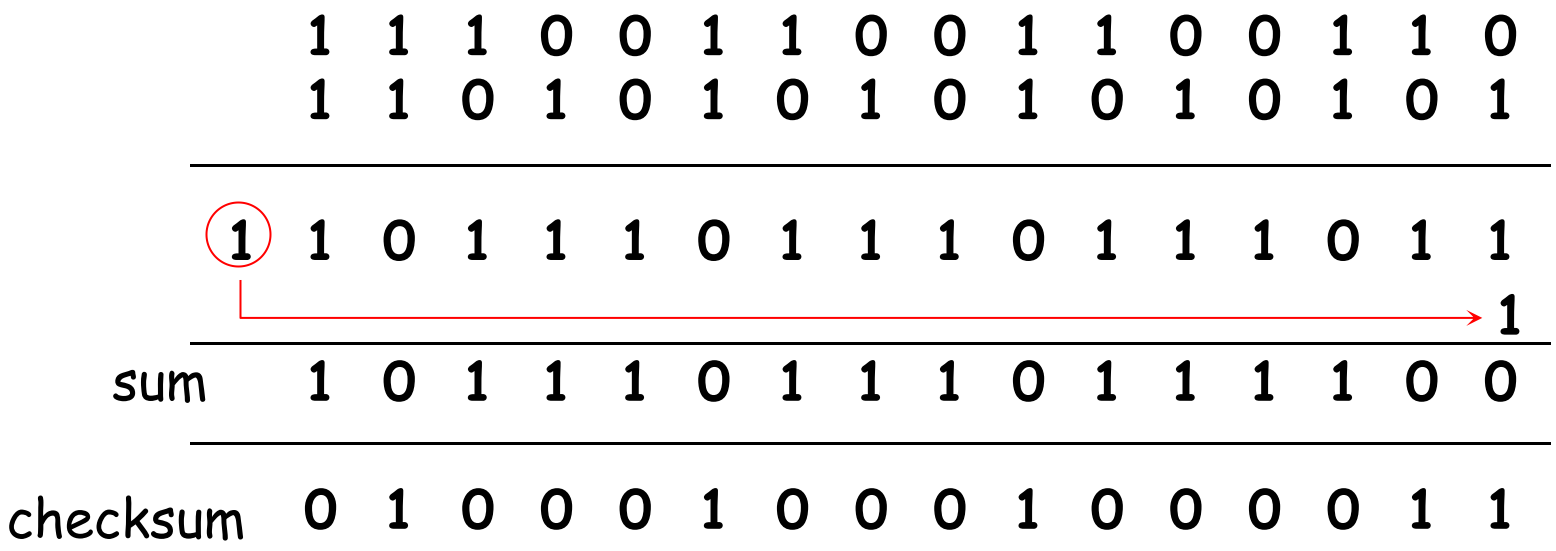
RFC 1071 Computing the Internet Checksum

- 采用**二进制反码运算（带进位的加法，进位加到个位)。教材有误（不是补码运算）**
  - 全1和全0分别对应着-0和+0
- 计算检验和字段：
  - 16比特的检验和字段初始化为全0
  - 消息以16-bit为一组(word)，按照反码运算将这些word相加，最后取相加的结果的反码，就是该消息的检验和字段取值
- 验证检验和是否正确：
  - 包含检验和字段在内的消息以16比特的word为单位，按照反码运算相加，如果为全1，则表示没有出错



验证 $\sum words + \overline{\sum words} = 1\cdots1$

TCP段(segment)格式：检验和算法




```
u_int16 cksum(u_int16 *buf, int count)
{
   register u_int32 sum = 0;
   while (count--)
   {
      sum += *buf++;
      if (sum & 0xFFFF0000)
      {
         /* carry occurred,so wrap around */
         sum &= 0xFFFF;
         sum++;
      }
   }
   return ~(sum & 0xFFFF);
}
```

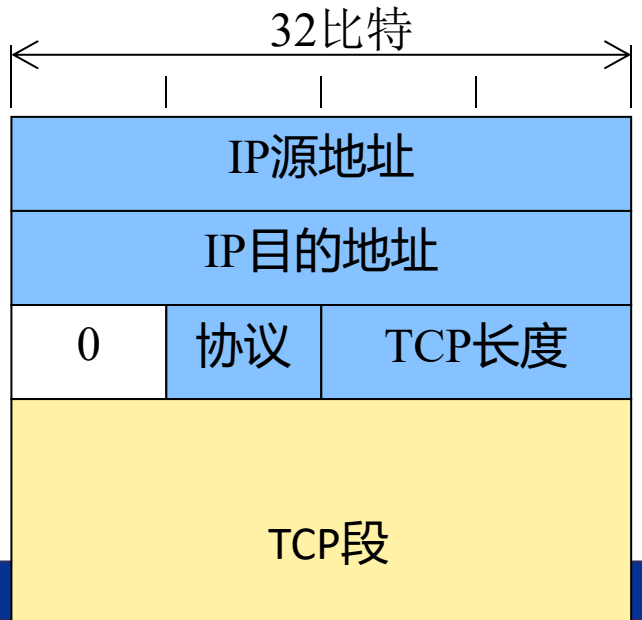
- 许多错无法检测，如果某些word增加了某个数量，而另外word减少同样的数量，检验和不变
- 更强大的检验和算法，提供接近于CRC的检错能力，但是轻量级计算：
  - Fletcher检验和: 8/16/32-bit checksum
  - Adler检验和： 32-bit checksum

TCP段(segment)格式：检验和算法

- 校验和：检测TCP段传输过程中差错（IP头部包含头部检验和
  - 计算检验和的消息由2部分组成：伪头部 + TCP段
  - 伪头部：
    - 源IP地址、目的IP地址、协议(8bit)、TCP段长度(16bit=总长度-IP头部长度)
    - 避免由于IP地址和协议号错误等造成的错误递交
    - 违反了协议分层的独立性

IP头部格式

TCP检验和计算使用的伪头部



TCP段(segment)格式

TCP段： 20字节的头部+最多40字节可选部分+用户数据

数据传输相关的字段

顺序号(32bit): 为每个字节编号

- 携带的用户数据第1个字节的顺序号
- 最后1个字节的顺序号=Seq + Data Len -1
- SYN=1时表示初始顺序号

ACK: 累加确认(32bit) ACK=1时有效

- 期待接收的下一字节的顺序号
- 除了最初的SYN段外，后续TCP段的ACK=1

接收窗口(16bit)

- 收到的数据放在接收缓冲区中；
- 应用程序从接收缓冲区中读取**已按序到达**的数据
- 接收者目前可用(空闲)的缓冲区大小

IP头部： **源IP地址 + 目的IP地址 + 协议号** + 总长度 + 头部长度 + IP头部检验和

32 位

源端口号									目的端口号
顺序号									
确认号(ACK置位时有效)									
数据偏移	保留(4b)	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	接收窗口
检验和									紧急指针(如果URG=1)
选项（0个或多个字节）									填充
用户数据 （可选）									

紧急指针(16bit): URG=1时有效。RFC6093不建议使用紧急数据功能

- 通知有紧急数据到来，顺序号+紧急指针给出了其对应的顺序号

- RFC 793: **紧急数据之后一个字节位置**
- RFC 1122: 紧急数据最后一个字节位置
- 紧急数据长度多少？大部分实现1字节

TCP段格式：TCP选项

- 每个选项的第一个字节为type，给出是哪种选项
  - EOL和NOP为单字节选项
  - 多字节选项一般采用TLV格式编码（8比特的Type + 8比特的Length + Value），注意Length指的是当前选项部分占的总长度
  - 必须支持EOL、NOP和MSS选项
- WSCALE:
  - 仅在SYN=1时使用，表示后续的TCP段的接收窗口的伸缩因子（SYN段本身不使用）
  - 取值[0, 14]，接收窗口= SEG.WND * $2^{伸缩因子}$ 。16比特的接收窗口扩展为30比特

仅在连接建立阶段(SYN=1)出现

名称	选项	1 byte	1byte		
EOL	End Of Options List	kind=0	选项部分结束		
NOP	No operation	kind=1	用于填充		
MSS	Maximum Segment Size	kind=2	len=4	最大分段大小	
WSCALE	Window Scale Factor	kind=3	len=3	移位位数(伸缩因子)	
SACK-permitted	发送方支持SACK	kind=4	len=2	协商是否采用SACK	
SACK	选择确认	kind=5	可变长	SACK块	
TSOPT	Timestamp	kind=8	len=10	时间戳值 TSval	时间戳回应 TSecr

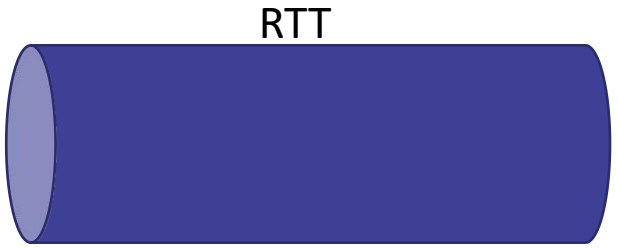
TCP段格式：TCP选项示例

- 连接建立和连接释放：SYN或FIN=1的TCP段
- MSS/WS/SACK_PERM都仅仅出现在SYN=1的TCP段中
- MSS为4个字节
- NOP + 3字节的WSS
- NOP + 2字节的SACK_PERM

Protocol	Length	Info
TCP	66	9695 → 443 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM=1
TCP	66	443 → 9695 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1340 WS=256 SACK_PERM=1
TCP	54	9695 → 443 [ACK] Seq=1 Ack=1 Win=262144 Len=0
TLSv1.2	290	Client Hello
TCP	1394	443 → 9695 [ACK] Seq=1 Ack=237 Win=524800 Len=1340 [TCP segment of a reassembled PDU]
TCP	1394	443 → 9695 [ACK] Seq=1341 Ack=237 Win=524800 Len=1340 [TCP segment of a reassembled PDU]
TCP	1394	443 → 9695 [ACK] Seq=2681 Ack=237 Win=524800 Len=1340 [TCP segment of a reassembled PDU]
TCP	1394	443 → 9695 [ACK] Seq=4021 Ack=237 Win=524800 Len=1340 [TCP segment of a reassembled PDU]
TCP	1394	443 → 9695 [ACK] Seq=5361 Ack=237 Win=524800 Len=1340 [TCP segment of a reassembled PDU]
TLSv1.2	588	Server Hello, Certificate, Certificate Status, Server Key Exchange, Server Hello Done
TCP	54	9695 → 443 [ACK] Seq=237 Ack=7235 Win=262144 Len=0
TLSv1.2	212	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
TLSv1.2	735	Application Data
TCP	60	443 → 9695 [ACK] Seq=7235 Ack=1076 Win=524032 Len=0
TLSv1.2	105	Change Cipher Spec, Encrypted Handshake Message
TCP	54	9695 → 443 [ACK] Seq=1076 Ack=7286 Win=261888 Len=0
TLSv1.2	218	Application Data
TCP	54	9695 → 443 [ACK] Seq=1076 Ack=7451 Win=261888 Len=0
TCP	54	9695 → 443 [FIN, ACK] Seq=1076 Ack=7451 Win=261888 Len=0
TCP	60	443 → 9695 [ACK] Seq=7451 Ack=1077 Win=524032 Len=0

```
Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operation (NOP), No-Operation (NOP), SACK permitted
  TCP Option - Maximum segment size: 1460 bytes
  TCP Option - No-Operation (NOP)
  TCP Option - Window scale: 8 (multiply by 256)
  TCP Option - No-Operation (NOP)
  TCP Option - No-Operation (NOP)
  TCP Option - SACK permitted
```

长肥网络LFN(Long Fat Networks)

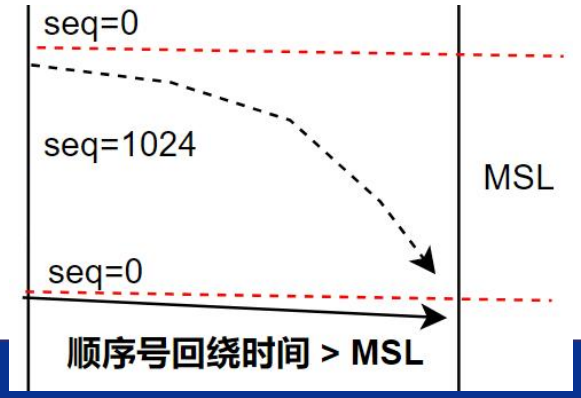


- RTT * Bandwidth:
  - 管道的容量，TCP发送者最多允许发送的尚未确认的数据大小
  - 如果其**远远超过10^5 比特或12500字节**，则称为长肥网络LFN，发音elephan(t)

100Mbps x 10 ms = 10^6 bit
= 122.07KB > 64 KB

- 接收窗口(16bit)： 意味着最大为65535字节
  - WSCALE选项允许接收窗口最大为$(2^{16}-1)2^{14}$ =1,073,725,440字节，大约1G字节
- 顺序号(32bit): 为每个字节编号
  - IP分组（封装了TCP段）有一定的生命周期**MSL(Maximum Segment Lifetime)**，建议为2分钟（120秒），在实践中经常假设为30秒
  - **顺序号回绕**：一个顺序号使用了，如果接下来传输了2^{32} 个字节的数据，则该顺序号又被再次使用
  - 顺序号回绕时，确保前一个顺序号已经不会出现在网络中了，即要求 **顺序号回绕时间 > MSL**
  - 时间戳选项TSOPT用于解决数据速率过快带来的顺序号回绕

MTU=1500



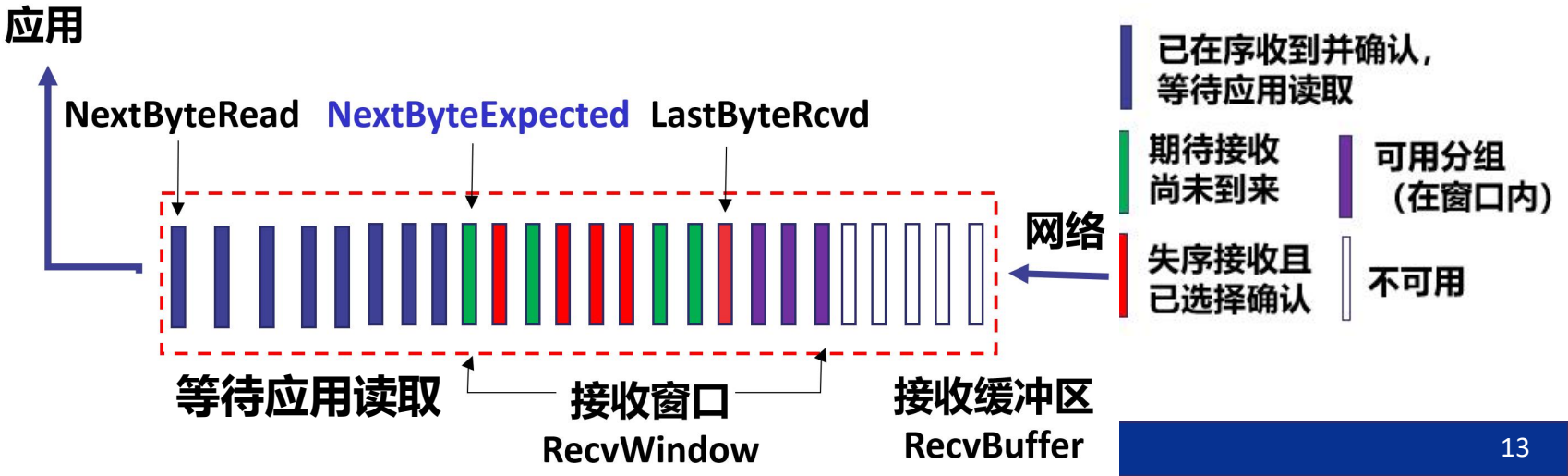
数据速率	顺序号回绕时间
1Mbps	9.8 hours
10Mbps	59 minutes
100Mbps	5.9 minutes
1Gbps	35.2 seconds
10Gbps	3.5 seconds

TCP流量控制：接收方

接收方采用累加确认，会缓存那些失序到达的TCP段，接收窗口大小可变

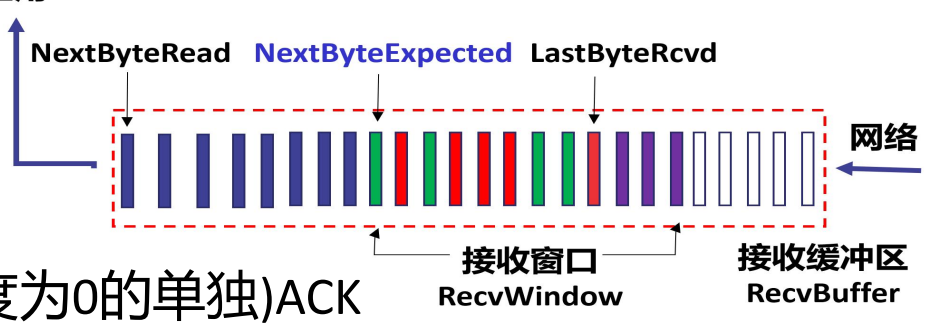
- RecvBuffer: 接收方缓冲区大小，保存收到的尚未被应用读取的数据或者失序到达但是在接收窗口中的数据。可通过setsockopt(SO_RCVBUF)设置最大值 。Linux的实现允许动态调整接收缓冲区大小
- NextByteExpected:期待接收的下一个顺序号，之前都已在序接收
- LastByteRcvd：收到的保存在接收缓冲区中的数据的最后1个字节顺序号，可能失序到达
- NextByteRead: 应用下次读取的数据的第1个字节顺序号
- **目前已经在序收到尚未被应用读取的数据**： NextByteExpected - NextByteRead
- 接收窗口：接收缓冲区的可用空间，用于缓存那些新到达的TCP段。通过ACK发送给发送方
- 接收缓冲区的限制： **RecvWindow <= RecvBuffer - (NextByteExpected - NextByteRead)**

- 接收窗口后沿:
  NextByteExpected + RecvWindow -1
- 在序收到数据时，更新接收窗口前沿，发送ACK
- 应用进程读走数据时，可用空间增加，足够大时，发送**窗口更新通知ACK**（ACK顺序号不变，但接收窗口改变）



TCP流量控制：接收方

拓展的内容

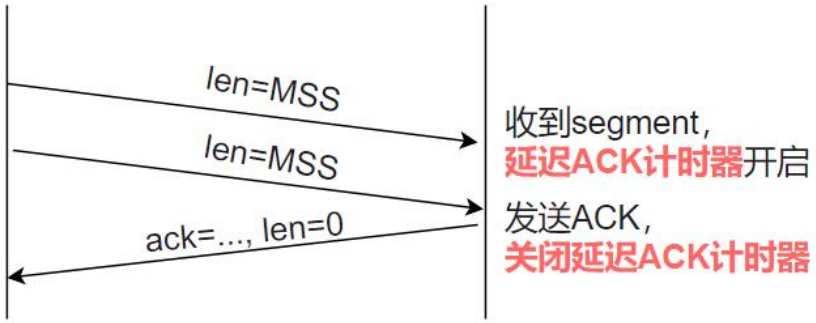


在收到一个携带了数据的TCP段时，什么时候发送(**累加**)ACK?

- 如果到来的数据在接收窗口左侧(重复)，则立即发送重复的(长度为0的单独)ACK
- 如果数据失序到达，填充了接收窗口中的空袭，则立即发送重复ACK（参见后面的**快速重传**）
- 如果到来数据填充了接收窗口空隙的最开始部分，则立即发送新的ACK（参见**快速重传**）
- 如果数据在序到达（缓冲区中目前没有失序到达的数据），则应该发送新的ACK
  - 数据传输是双向的，可捎带确认，但具体实现?
  - 延迟ACK（Delayed ACK)：延迟一段时间，反方向是否有数据传输以**捎带确认**
    - 最多等待500ms(早期的规范），现在一般为200ms
    - TCP的数据传输是ACK驱动的：根据接收者发送的ACK调整发送窗口、拥塞窗口
      - 如果发送者连续发送多个携带了MSS字节用户数据的TCP段，则每收到两个在序到达的TCP段时至少发送一个ACK。Linux实现：收到数据超过MSS时，下一个TCP段到来将立即ACK
    - 标准的socket接口无法关闭延迟ACK

拓展：Linux的延迟ACK实现：可setsockopt(TCP_QUICKACK) 暂时进入QUICKACK模式

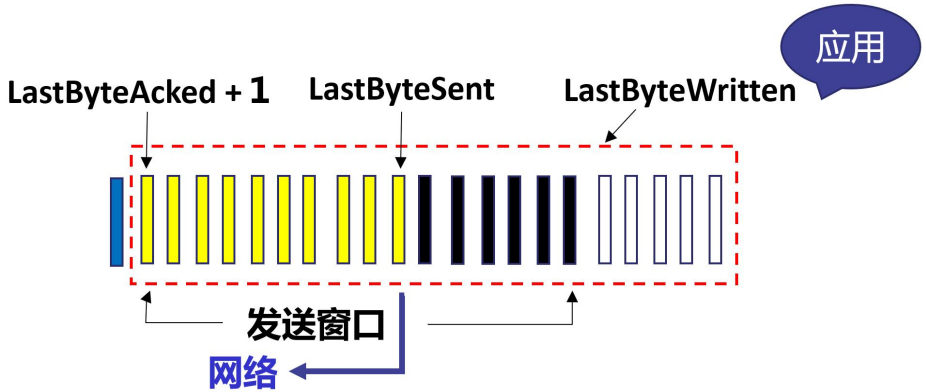
- 延迟ACK的时间最小可为40ms
- 可暂时进入QUICKACK模式，立即发送ACK。根据收发情况可能会退出该模式
  - 检测是否为交互式(pingpong)，如果是，可延迟ACK以捎带确认：根据上次收到TCP数据段的时刻与发送TCP数据段的时刻判断



TCP流量控制：发送方

- 发送方由ACK驱动，收到ACK后滑动发送窗口
- SendBuffer：发送缓冲区大小，存放来自于网络应用的待发送的用户数据。可通过setsockopt设置最大值。 Linux的实现允许动态调整发送缓冲区大小
- LastByteAcked: 发送方发送的已被接收者确认的数据中最后1个字节的顺序号
- LastByteSent：已经发送的数据中最后一个字节的顺序号
- LastByteWritten: 应用进程写入的数据中最后一个字节的顺序号
- 缓存在发送缓冲区的数据长度： LastByteWritten – LastByteAcked
- 可以继续写的字节数： SendBuffer - (LastByteWritten – LastByteAcked)
- **已经发送尚未确认**(outstanding或in-flight)的数据长度： LastByteSent – LastByteAcked
- **可用窗口**（还可以继续发送的字节数）：

  SendWindow - (LastByteSent – LastByteAcked)



TCP流量控制：发送方

应用进程来的数据:

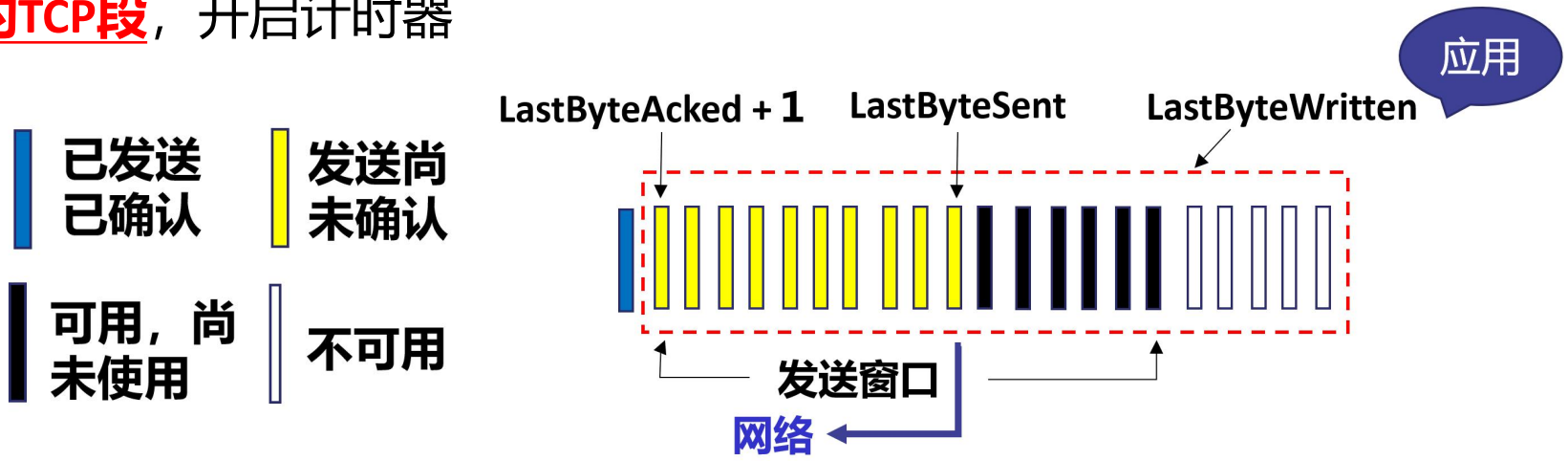
- 如果发送缓冲区允许，放入缓冲区，分配顺序号
- 如果发送窗口允许，发送数据，更新LastByteSent。如果计时器没有开启，则开启计时器

ACK: seg.ack和seg.win

- 如果为**新的ACK**，即确认了新数据的ACK： seg.ack > LastByteAck + 1 或**窗口更新ACK：** seg.ack == LastByteAck + 1，且seg.win比最近收到的接收窗口更大
  - **更新发送窗口**：LastByteAcked = seg.ack - 1，SendWindow = seg.win
  - 如果发送窗口允许且有未发送数据，发送并开启计时器
- 如果为重复ACK→ 见快速重传： 3个连续的重复ACK后重传该丢失的分组

超时:

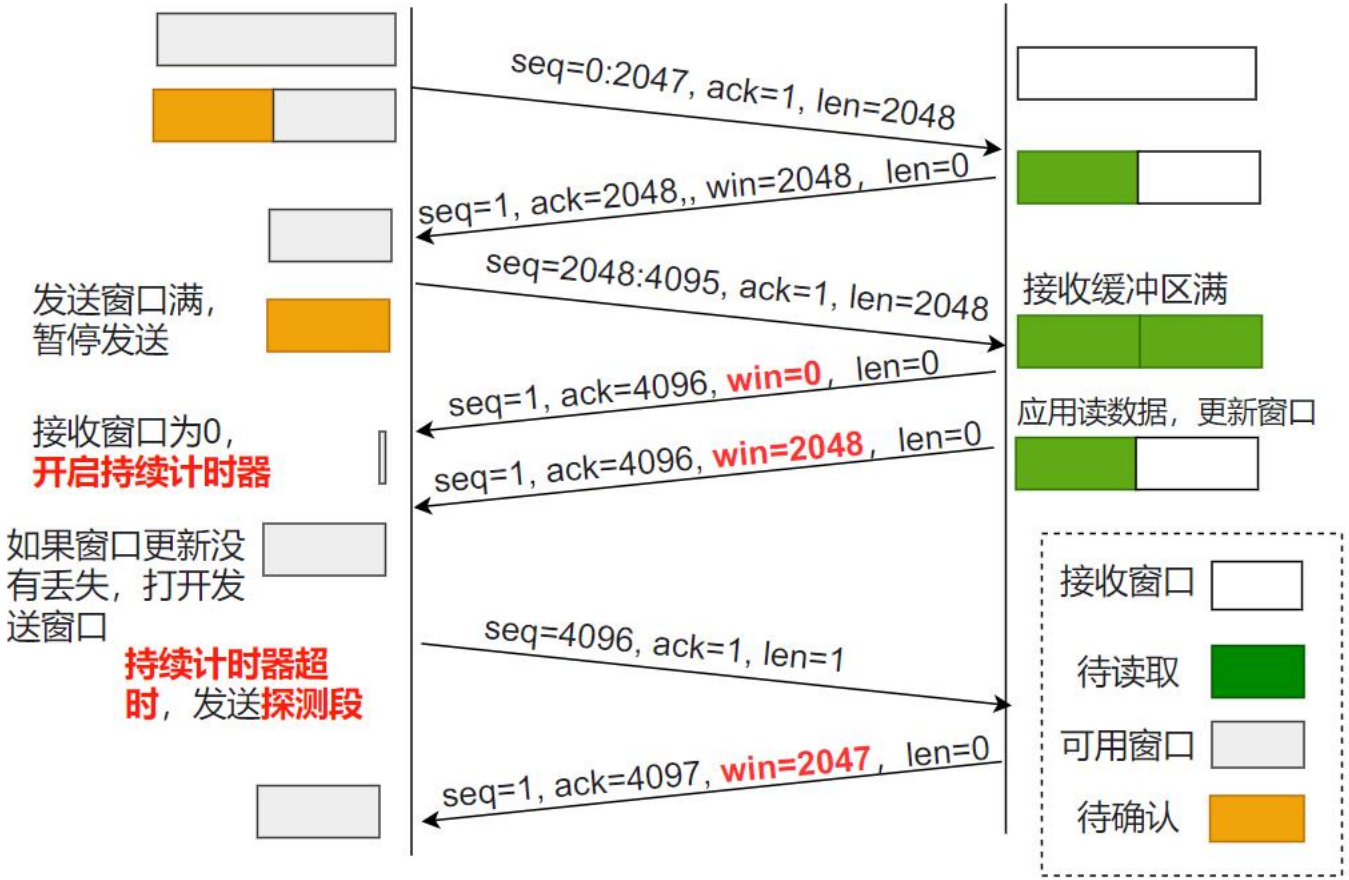
- **重传第一个未确认的TCP段**，开启计时器



TCP流量控制：接收窗口为0

如果接收窗口为0

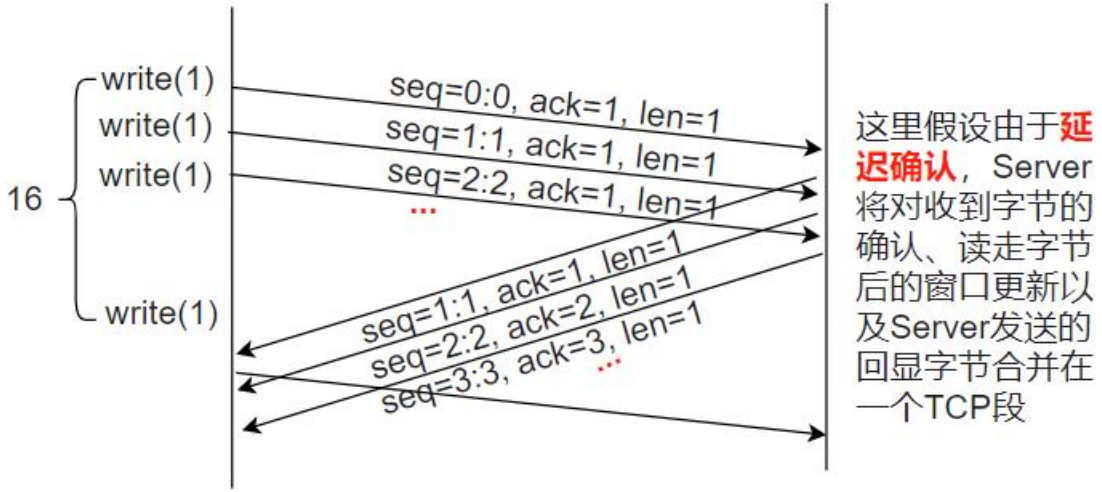
- 发送者尽管有数据，但是不允许发送
- 当接收窗口打开（变为非0）后，发送窗口更新ACK，但是如果该ACK丢失?
  - 发送方等待，接收方不会重传窗口更新ACK，也不会主动发送ACK，出现死锁
  - 除非应用程序读走更多的数据导致接收窗口再次变化，发送新的窗口更新ACK
- 发送者发现接收窗口为0时开启持续(persistence)计时器
  - 如果接收窗口打开，则停止该计时器
  - 超时时发送probe分组，如果接收窗口仍然为0，重启持续计时器
  - probe分组：
    - 发送长度至少为1的新TCP段，seq=LastByteAcked+1，或者重传(重复的)TCP段
  - 持续计时器初始为当前重传超时，重启时翻倍，直到60秒为止

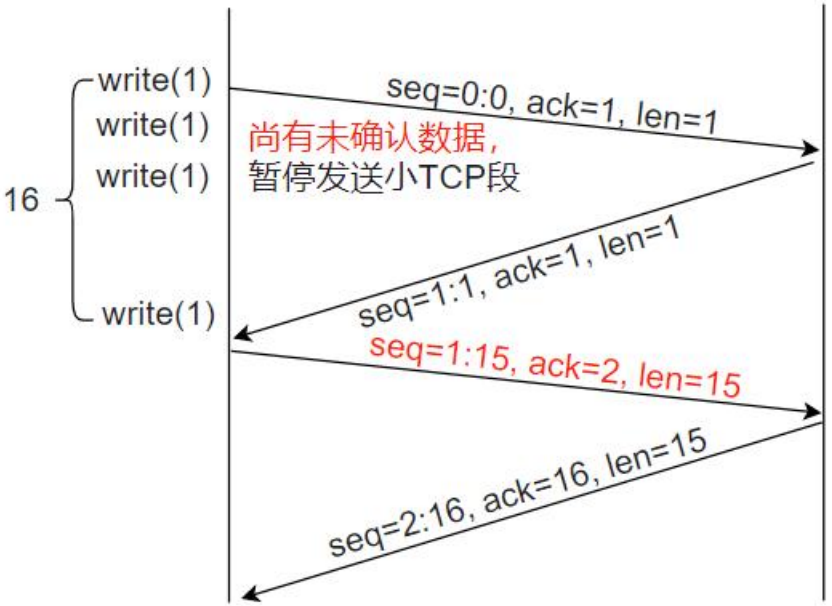


TCP流量控制：小分组（Tinygram）

由于发送者产生大量小数据，导致许多小的TCP数据段出现

- 大量小的TCP段浪费了带宽，也浪费了CPU
- **发送方采用Nagle算法**(RFC 896)：在保证一定交互性的同时尽可能发送更多的数据
  - 发送小分组时，类似于停等协议：某个方向只允许有一个小分组在传输，在所有的ACK回来之前，暂时停止发送
  - 如果连接空闲（即所有发送的分组都已经确认），可以马上发送一个小分组
  - 否则（由于已经有ACK在路途，可以）等待积累足够多的数据



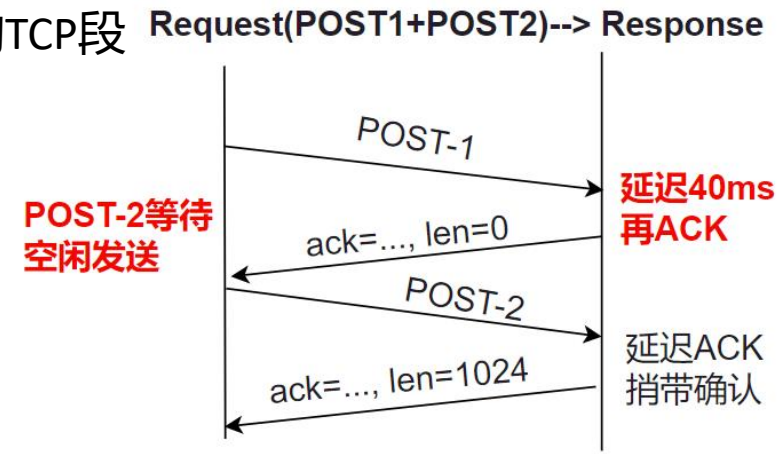


TCP流量控制：小分组（Tinygram）

- 可不采用Nagle算法，setsockopt(TCP_NODELAY)
  - 带宽充裕时
  - 某些交互式应用(如网络游戏、X窗口协议)，应尽可能快地发送数据
- 拓展：Linux系统支持TCP_CORK选项，该选项设置时表示不允许发送小的TCP段，而清除TCP_CORK选项时，允许发送小的TCP段
  - 首先开启TCP_CORK选项
  - 多次小的数据的发送
  - 最后清除TCP_CORK选项

```
sock.setsockopt(socket.SOL_TCP, socket.TCP_NODELAY, True)
sock.setsockopt(socket.SOL_TCP, socket.TCP_CORK, True)
sock.send(data0)
sock.send(data1)
sock.setsockopt(socket.SOL_TCP, socket.TCP_CORK, False)
```

- 拓展：Nagle算法（要等待ACK）与Delayed ACK（延迟ACK）结合可能带来问题
  - 考虑request-response协议，但是client方发送的request时分成了两个小的TCP段
  - 根据Nagle算法，POST-2要等待连接空闲后再发送
  - 收到POST-1之后，请求不完整，无法捎带确认，40ms后超时发送ACK
  - 客户方收到ACK后开始发送POST-2，服务方收到后，延迟ACK
  - 很快服务方对请求进行响应，同时进行捎带确认



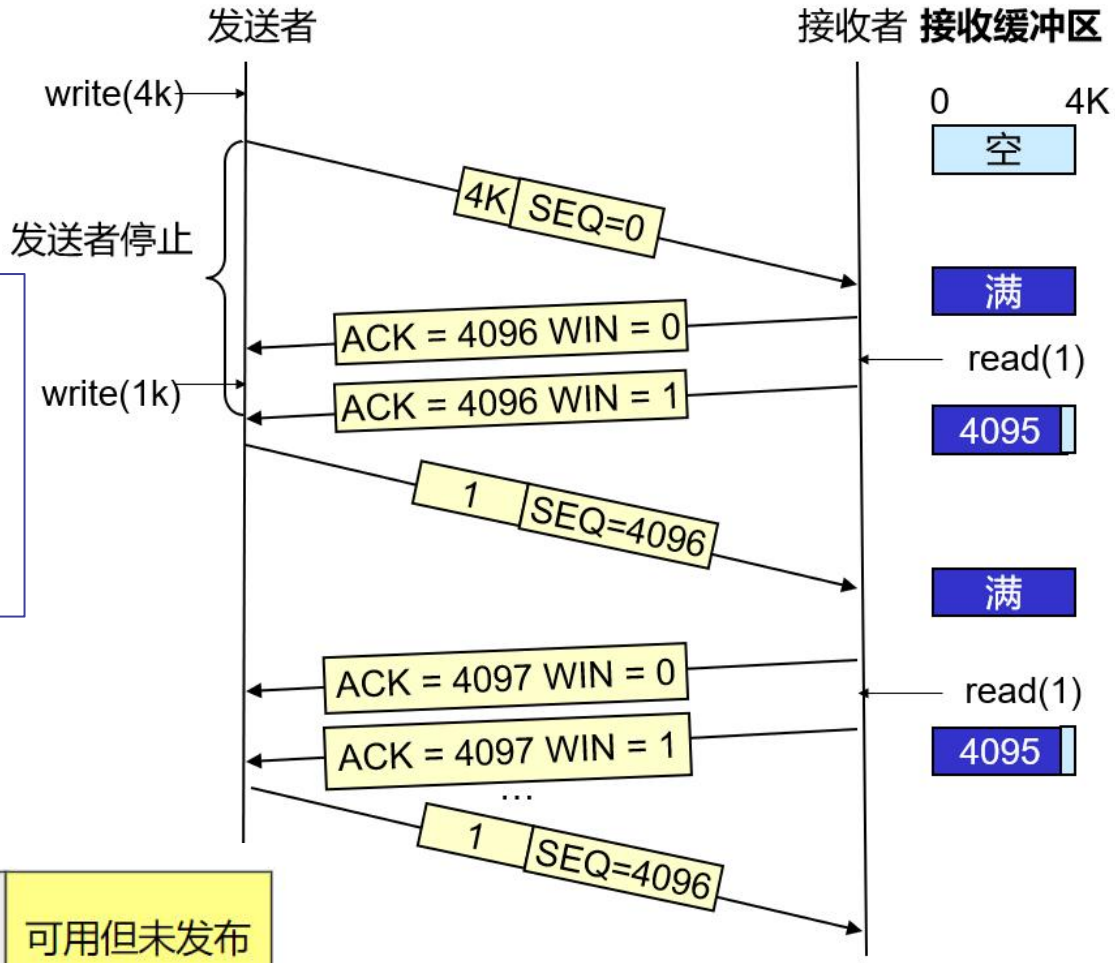
TCP流量控制：傻瓜窗口症状(Silly Window Syndrome)

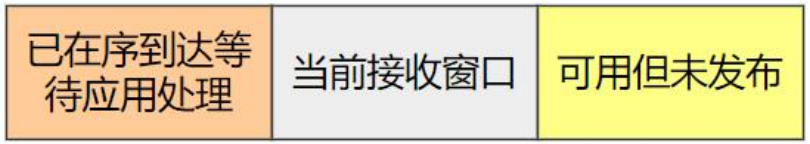
由于接收方读取数据比较慢导致尽管有大量数据，但是由于发送窗口的限制，只能发送许多小的TCP数据段，也称为**糊涂窗口综合症**

- 接收方应用**每次读取少量数据**，有更多的缓冲区时(即窗口右沿滑动）马上发送窗口更新ACK通知发送者
- 发送者发现发送窗口右沿往前滑动了，可以发送，但只能发送小的TCP段（尽管有大量数据等待发送）

发送方 延迟发送，直到可发送足够大的TCP段或连接空闲：
- 可以发送一个MSS大小的段
- 可以发送的段足够大（最大窗口的一半）
- 连接目前空闲（所有ACK都回来） √
- 没有采用delay选项，并且有数据要发送 √ 关闭Nagle算法

接收方 接收窗口变化时延迟发送窗口更新ACK：
- 不要发送较小的窗口增长通知
- 直到可用窗口达到相当大的程度：
  min(MSS,RecvBuffer/2)





TCP流量控制：快速重传(Fast Retransmit)

超时:

- 重传第一个未确认的TCP段，开启计时器

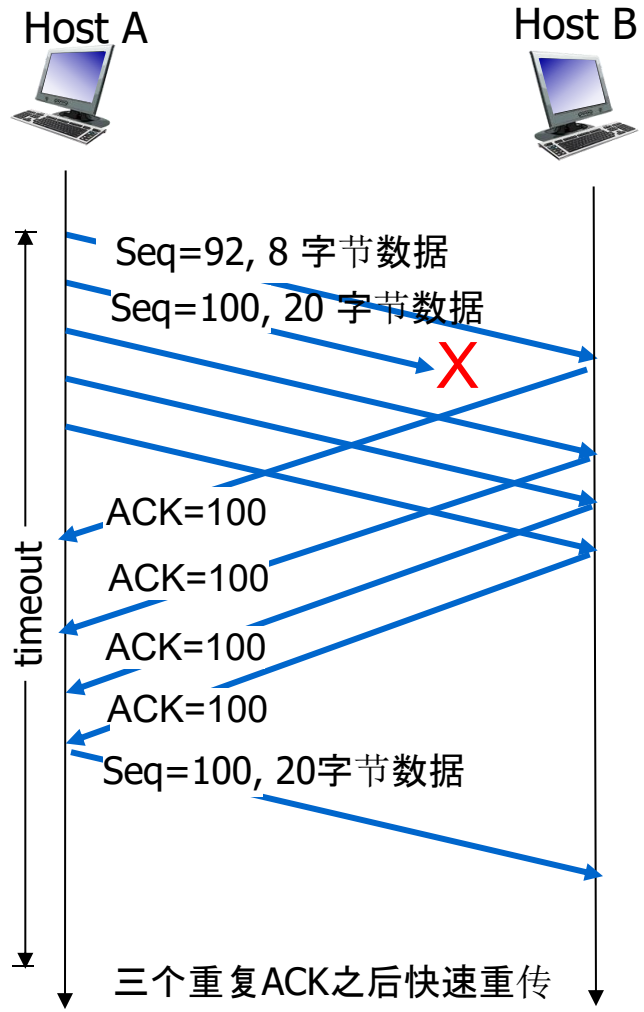
- 发送者重传超时一般比实际的RTT要长一些：等待超时才能重传丢失分组
- 接收者：收到失序TCP段也要发送重复ACK（最后一个收到的按序到达的TCP的ACK)
- 发送者经常连续发送多个TCP段，分组丢失时会触发多个重复ACK。收到一个重复的ACK，可能是：
- 被确认(期待接收)的TCP段后面的TCP段被暂时延迟，但最终会到达，尽管是失序到达：无需重传
- 该TCP段丢失：一个重复的ACK可以看作是一个预警，它告诉源端一个TCP段已经丢失了而必须重传

发送方：引入dupack：记录最近收到的重复ACK个数

如果到来为新的ACK：dupack = 0

如果到来为重复ACK：dupack++

TCP发送方收到3个重复ACK(即共收到同一报文段的4个ACK)：随后的TCP段已丢失的可能性就很大，立即重传该丢失的TCP段



TCP流量控制总结

GBN(Go back N)

- 接收者只允许顺序接收，即接收窗口=1
- 接收者发送**累加确认**
- 计时器超时时，重传**所有未确认分组**

TCP

- 接收方允许失序接收，**接收窗口可变**
- 接收者发送**累加确认**
- 接收方会发送**窗口更新ACK**
- 计时器超时时，重传**第1个未确认分组**
- **快速重传：3个重复ACK时快速重传第1个未确认分组**
- 性能方面的增强：避免小TCP段出现
  - 延时ACK以支持捎带确认和ACK压缩
  - 可用窗口足够大时才发送窗口更新ACK
  - Nagle算法：连接空闲时允许发送小的TCP段

选择重传(Selective Repeat)

- 接收方允许失序接收，即接收窗口>1

实现一

- 接收者对每个到达(包括**失序到达**)分组发送**单独ACK**
- 发送者为每个未确认分组各维护一个计时器，超时时，仅仅重传**该超时的分组**

实现二

- 采用**累加确认**
- 接收者可以发送**NAK**，请求**重传指定分组n**
- 每个分组对应一个计时器，超时时**重传该分组**

教材目录

3.3 数据链路协议：可靠数据传输

- 停等协议
- 滑动窗口协议：GBN和选择重传

6.2.1 TCP概述

6.2.2 TCP段格式

6.2.4 TCP可靠传输

6.2.5 TCP流量控制

TCP可靠数据传输

6.2.3 TCP连接管理：连接建立和连接释放

6.2.7 TCP计时器

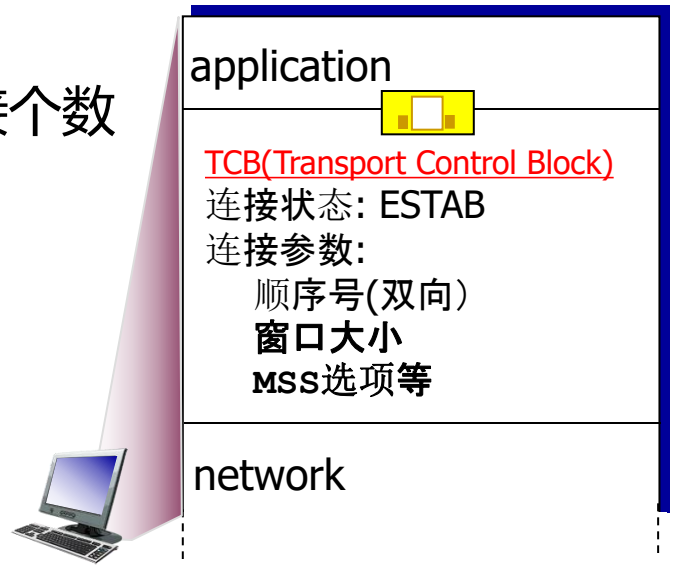
6.2.6 拥塞控制

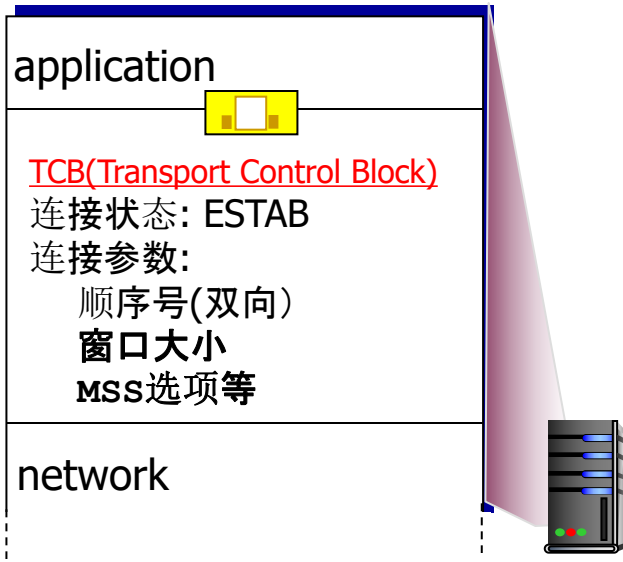
TCP连接管理：连接建立

- 数据传输之前首先要建立连接
  - 互相通知对方要求建立连接
  - 协商选项
  - 确定**数据传输所需要的参数(初始顺序号)**，分配资源

TCP连接建立是一个不对称的过程

- 主动方式的客户方(connect)向被动方式(listen)打开的服务方发送连接建立请求
- listen(backlog=0)
  - backlog: 已建立好但没有accept的连接个数






```
sock.connect((host, port))
```

```
sock.listen(5)
data_sock, addr = sock.accept()
```

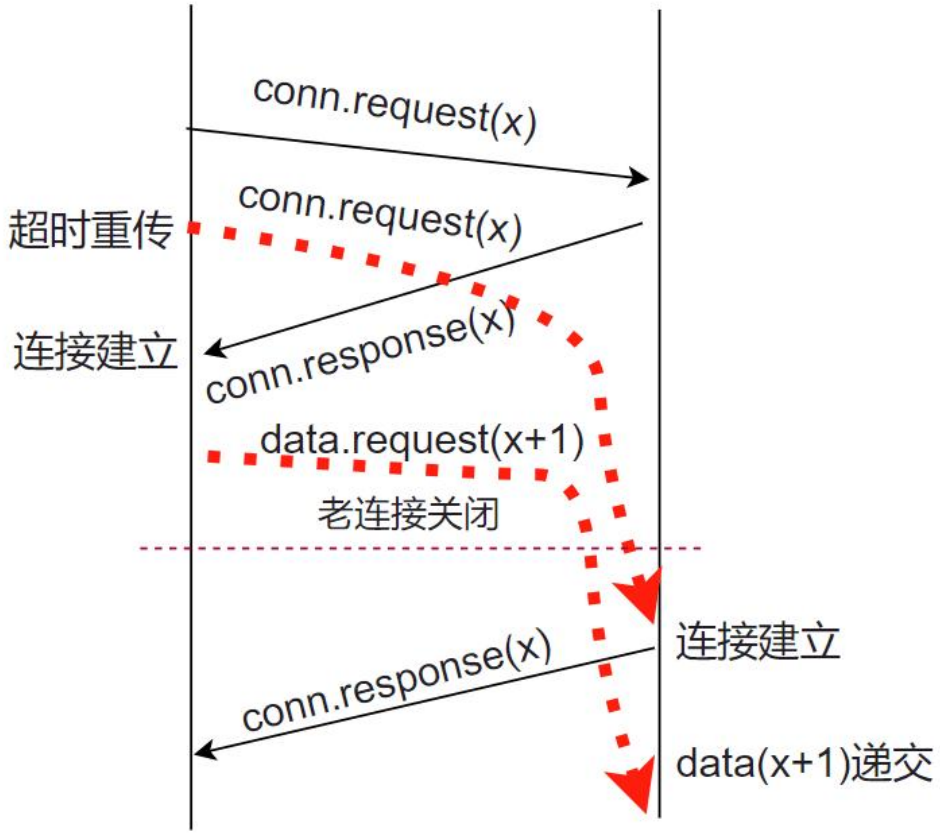
TCP连接管理：连接建立

下层为尽力递交的IP协议，要考虑：

- IP分组可能丢失、延迟、失序到达
- 前一次连接的分组可能会在新的连接期间到达，要避免前一次连接中的残留分组使用的顺序号与新连接正在采用的顺序号重叠
- 系统可能崩溃重启，无法记住前面连接使用的顺序号

连接建立如果采用两次握手过程，无法在IP层之上正常工作

- client发送连接建立请求(选择初始顺序号x)，服务方给以回应
- 老连接在建立连接、数据传输阶段都可能由于没有收到ACK而超时重传，这些分组可能在网络中游荡
- 来自老连接上那些还在网络中的conn.request、data.request等分组姗姗来迟时，导致连接建立，并且接受之前的重复数据

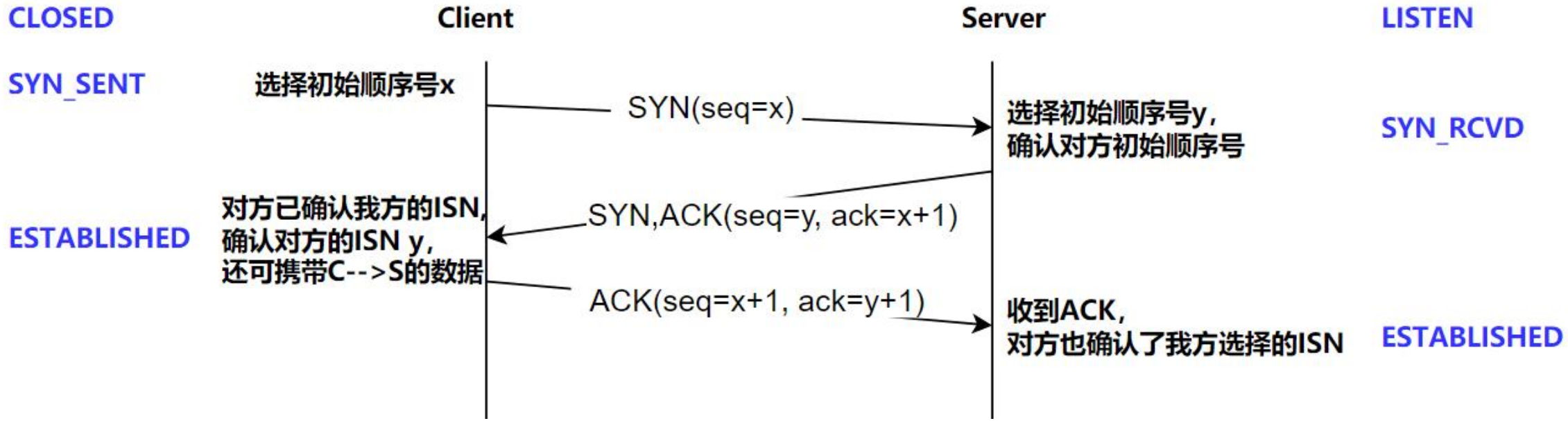


TCP连接建立：三次握手过程

连接建立的过程：

- 互相通知对方要求建立连接；协商选项（MSS/WSCALE/SACK)等
- 确定数据传输所需要的参数，最为关键是**同步初始顺序号ISN**

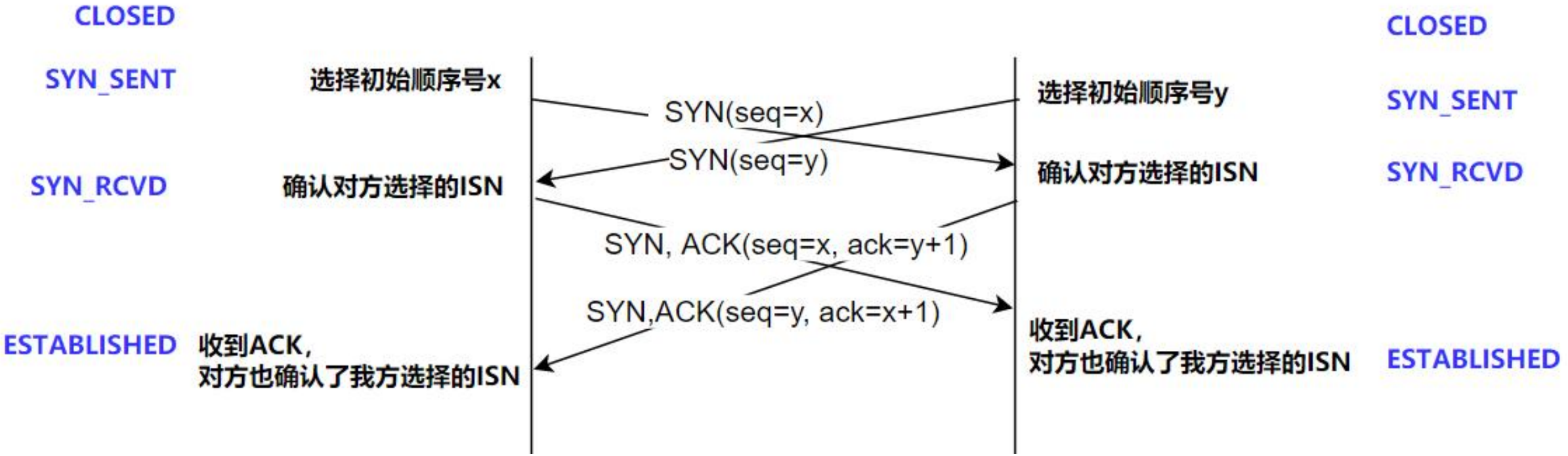
没有ACK标志的SYN为连接建立请求，之后的TCP段ACK都会置位。防火墙可针对此设定相应过滤规则



连接建立状态：已经互相同步初始顺序号
Cllient: 发送ISN=x，收到了对方的ACK(x+1)
Server: 收到了对方的ISN=y，发送了ACK (y+1)

TCP连接建立：三次握手过程

- 两台主机同时想在相同的套接字之间建立一条TCP连接而发生冲突，也可以正常工作

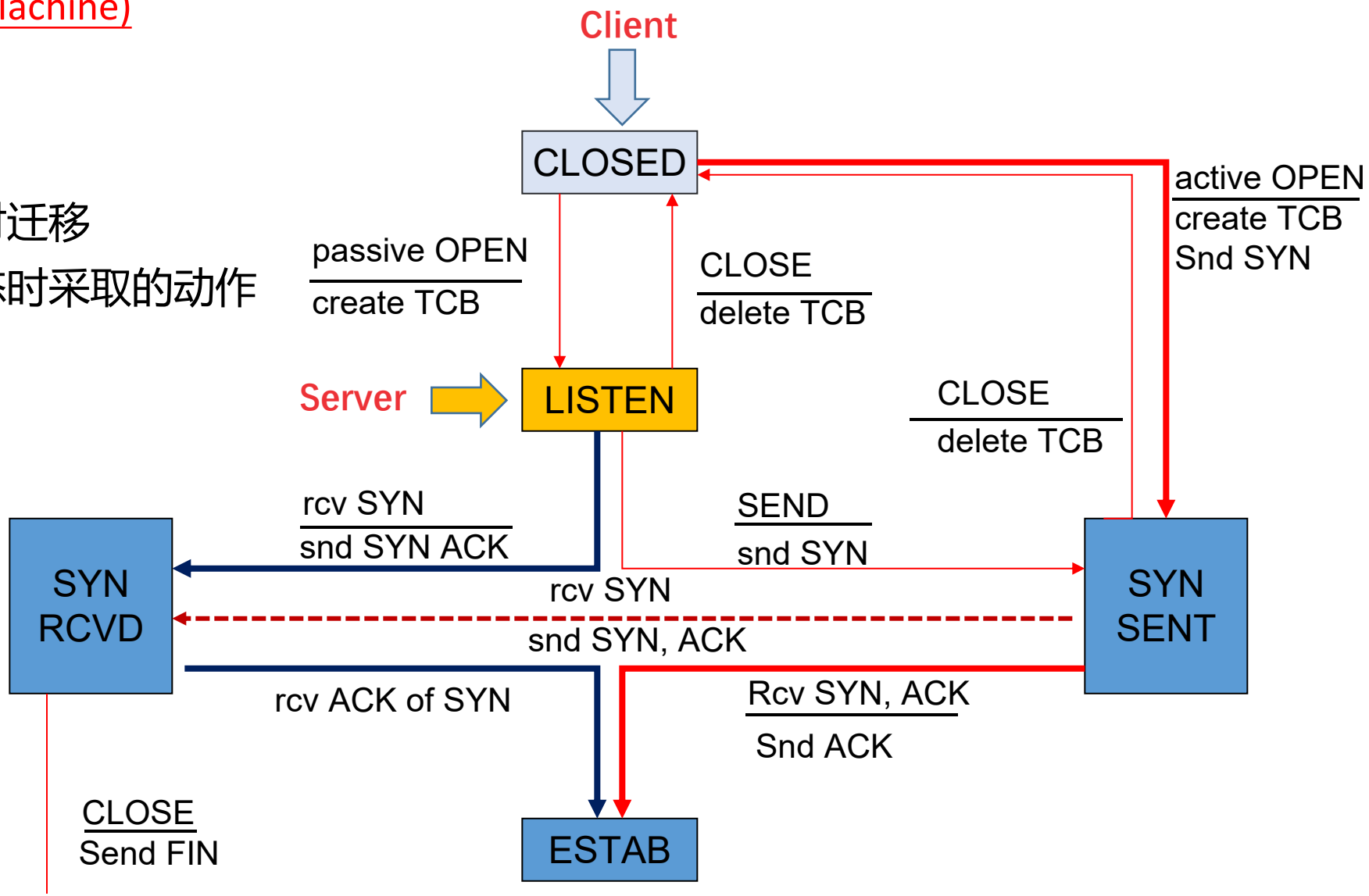


连接建立状态：已经互相同步初始顺序号
Cllient: 发送ISN=x，收到了对方的ACK(x+1)
Server: 收到了对方的ISN=y，发送了ACK (y+1)

TCP连接建立：状态转换图

有限状态自动机(Finite State Machine)

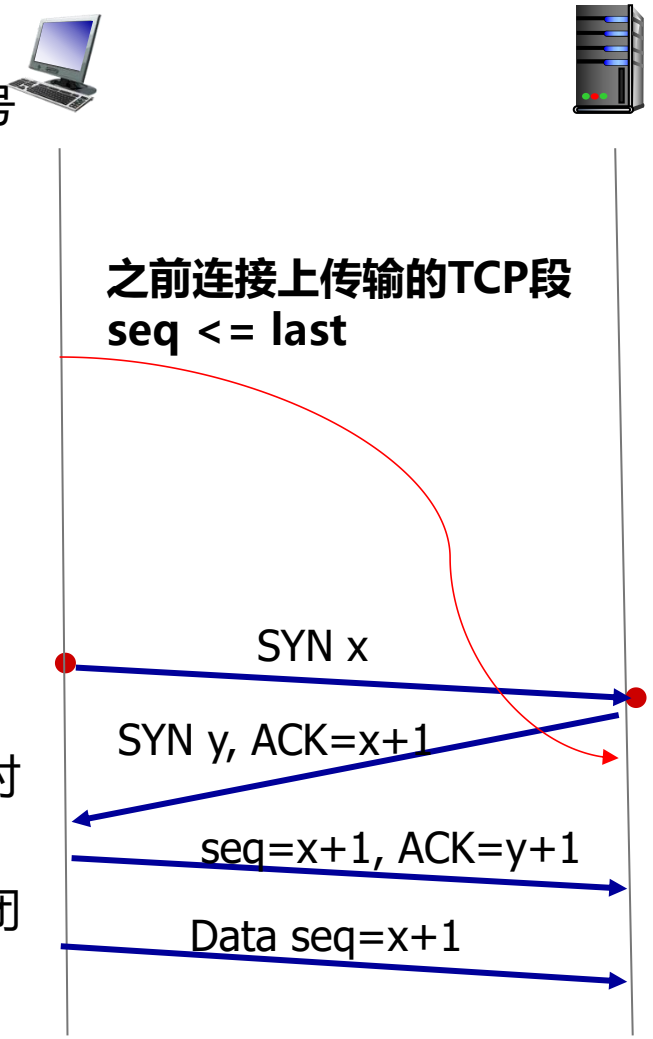
- 有限状态
- 状态之间的迁移：
  - 条件：满足相应条件时迁移
  - 动作：迁移到另一状态时采取的动作



TCP连接建立：初始顺序号

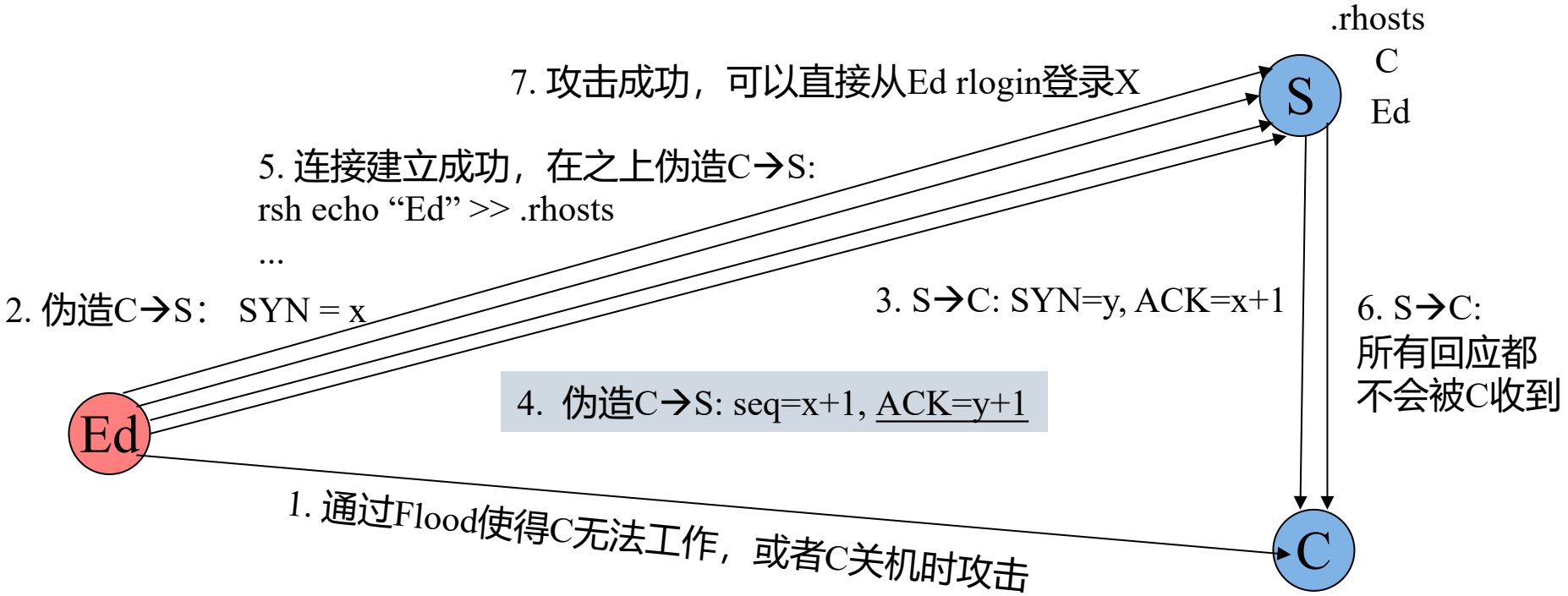
为什么要同步初始顺序号ISN? 而不是取固定的值，比如从0开始

- 前一次连接(出现过超时重传时）的分组可能会在新的连接期间到达，这些分组的顺序号可能与新连接正在采用的顺序号重叠
- 让主机记录之前的老连接使用过的顺序号?
  - 服务器有许多client连接时，开销太大
  - 系统可能崩溃重启，无法记住前面连接使用的顺序号
    - TCP段有一个最大生命期MSL(Maximum Segment Life)
      - RFC 793建议为2分钟（120秒），在实践中经常假设为30秒
    - 崩溃重启时，等待MSL时间，保证老连接的分组不会在新连接出现
- TCP采用三次握手过程，互相同步选择的(非固定)的初始顺序号
  - 老连接来的分组(顺序号和确认号）与新连接正在使用的合法顺序号范围正好匹配时才会有干扰
  - 连接释放时，主动关闭的一方将进入TIME-WAIT状态，等待2*MSL时间后才真正关闭
    - SYN请求只有在其**初始顺序号超过老连接的最后顺序号时**才会接受
  - 崩溃重启时TCP实现一般不会等待MSL。源端口动态选择，正好与崩溃前的TCP连接一致的几率不高；实践中分组的生命期要小得多；重启的时间一般已经超过了生命期



TCP连接建立：初始顺序号

- RFC793采用基于时钟的单调递增的ISN生成器：全局计数器值，每隔很短间隔 (4μs)加一，新连接使用计数器的当前值。但是ISN选取有规律带来IP Spoofing攻击
- 拓展：IP伪装攻击(IP Spoofing Attack):
  - 假设server X基于IP地址来信任client
  - 攻击者如果能够猜测到server所选取的ISN，伪造成client的身份完成连接建立的三次握手过程



- 攻击者一般来自于外部，配置防火墙，过滤掉那些可能伪造身份的SYN请求
- RFC 2827: ingress filtering, ，从某个接口收到的IP分组的源地址是否合法(不应该是内部网络的地址）？

TCP连接建立：初始顺序号

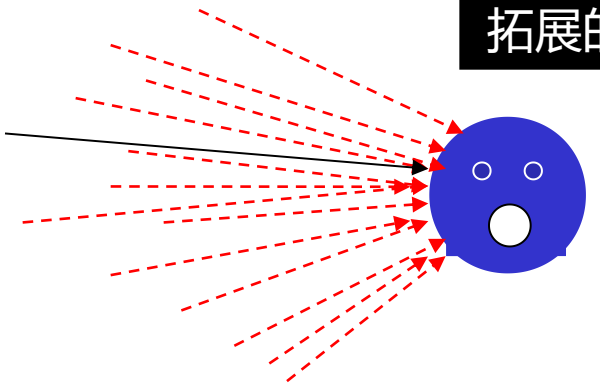
拓展的内容

- **源端口随机选取**
- **初始顺序号随机选取。**但是：
  - 处于TIME-WAIT状态时，收到的SYN请求只有在其**初始顺序号超过老连接的最后顺序号时**才会接受
- RFC 6528：避免IP伪装攻击，要求攻击者无法猜测到被攻击者选取的初始顺序号是多少
  - 源IP地址+端口号、目的IP地址+端口号标识一条TCP连接
  - 每个TCP连接拥有自身的顺序号空间
  - 假设攻击者无法监听待攻击的TCP连接上传输的分组，否则可以直接看到被攻击者选取的初始顺序号
  - 初始顺序号的基准值：基于TCP连接的标识以及某个密钥进行选取，攻击者无法监听和猜测到
  - 初始顺序号=基准值 + ISN生成器的当前值，保证初始顺序号递增

ISN = Counter + Hash(localip + port，remoteip+port, secretkey)

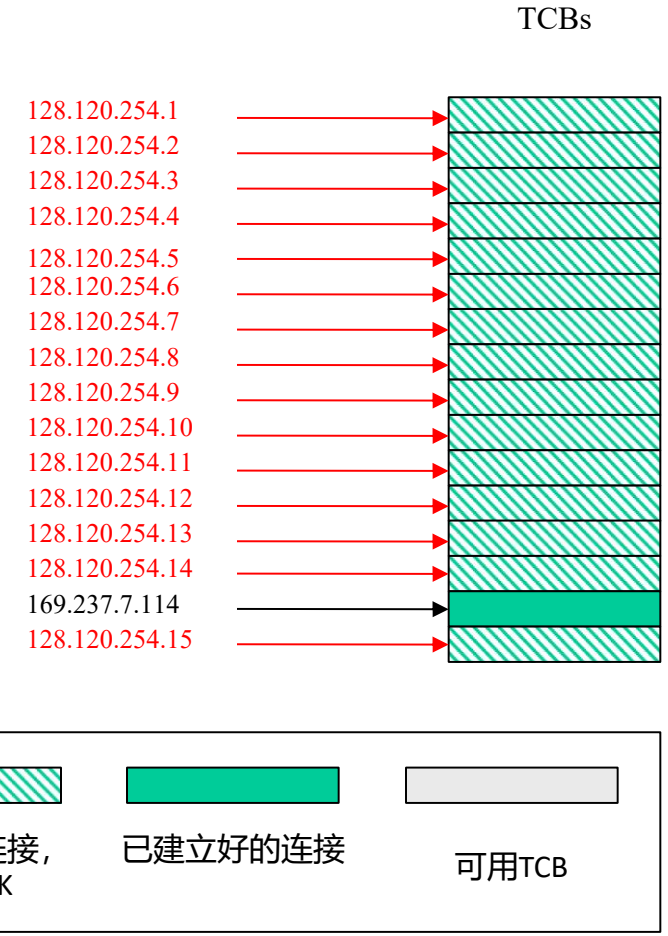
TCP SYN Flooding Attack

拓展的内容



- **RFC 4987 TCP SYN Flooding Attacks and Common Mitigations**
- TCP实现中的主要问题：连接建立过程中占用资源
  - 收到SYN时, 分配TCP Control Block (TCB)
    - > 280 bytes
    - 包括TCP流标识, 计时器, 顺序号, 流量控制变量, 紧急数据, 协商的选项(MSS等)
  - 这些Half-open TCB只有在超时后才会被移走
  - half-open连接的个数一般是有限的，

  sysctl net.ipv4.tcp_max_syn_backlog 返回2048
  - listen(backlog)：已经建立好但是没有accept的连接个数
- 伪造多个SYN请求，TCB的资源耗尽， 拒绝新的正常的连接建立请求

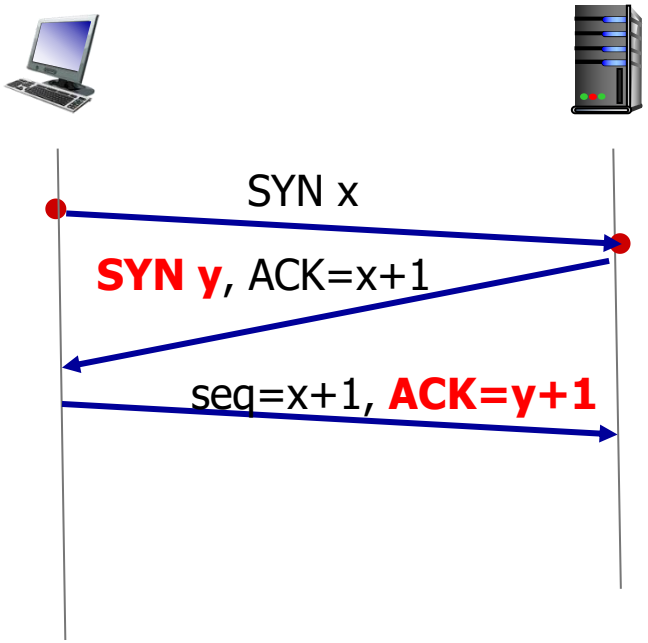


SYN Cookies

拓展的内容

32	5 bits	3 bits	
t mod 32	MSS	Cookie=HMAC(t, N_s, SIP, SPort, DIP, DPort)	

- 服务方在收到SYN请求时并不分配TCB
  - 计时器、流量控制、紧急数据等信息在连接完全建立之后才有用
  - TCP流ID在三次握手过程中传递
  - 对方的初始顺序号N_s可以从三次握手过程中最后的ACK中获得
  - 自己选取的初始顺序号可以从最后一个ACK中获得，但是需要验证该ACK确认的是服务方选取的初始顺序号
  - 最初SYN协商的TCP选项是什么?
- 服务方维护一个缓慢增加的时钟tick=time( ) >> 6，即每64秒增加1个tick
- Server收到SYN时，并不分配TCB，而是发送SYN-ACK Cookie
  - cookie = f(src addr, src port, dest addr, dest port, N_s, rand)
  - 初始顺序号由时钟的低5位、MSS选项和Cookie组成
  - SYN Cookie**仅支持MSS选项**➔ 操作系统一般缺省关闭SYN Cookie
- 服务方收到Client发送的ACK后验证该ACK是否合法
  - 当前的时钟tick与ACK的高5位还原为发送SYN-ACK时的tick，进行同样运算
  - 如果合法，则建立连接，分配TCB
- SYN Flooding的攻击者很难了解到服务方选取的初始顺序号，无法伪装发送最后的ACK

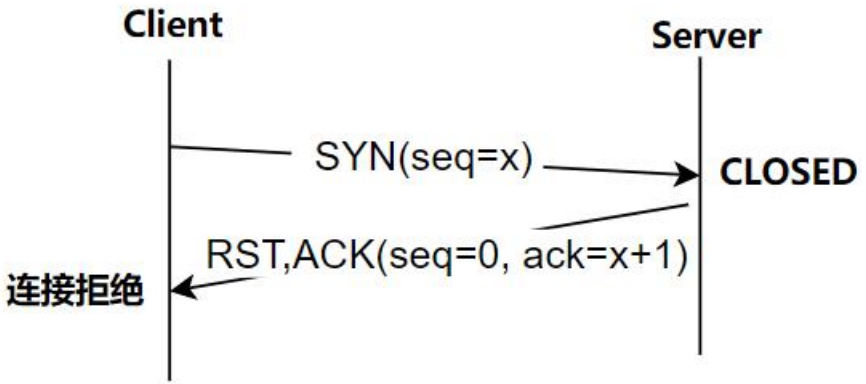


这里只是描述了一种可能的SYN Cookie生成和验证方法

TCP连接：重置(reset)连接

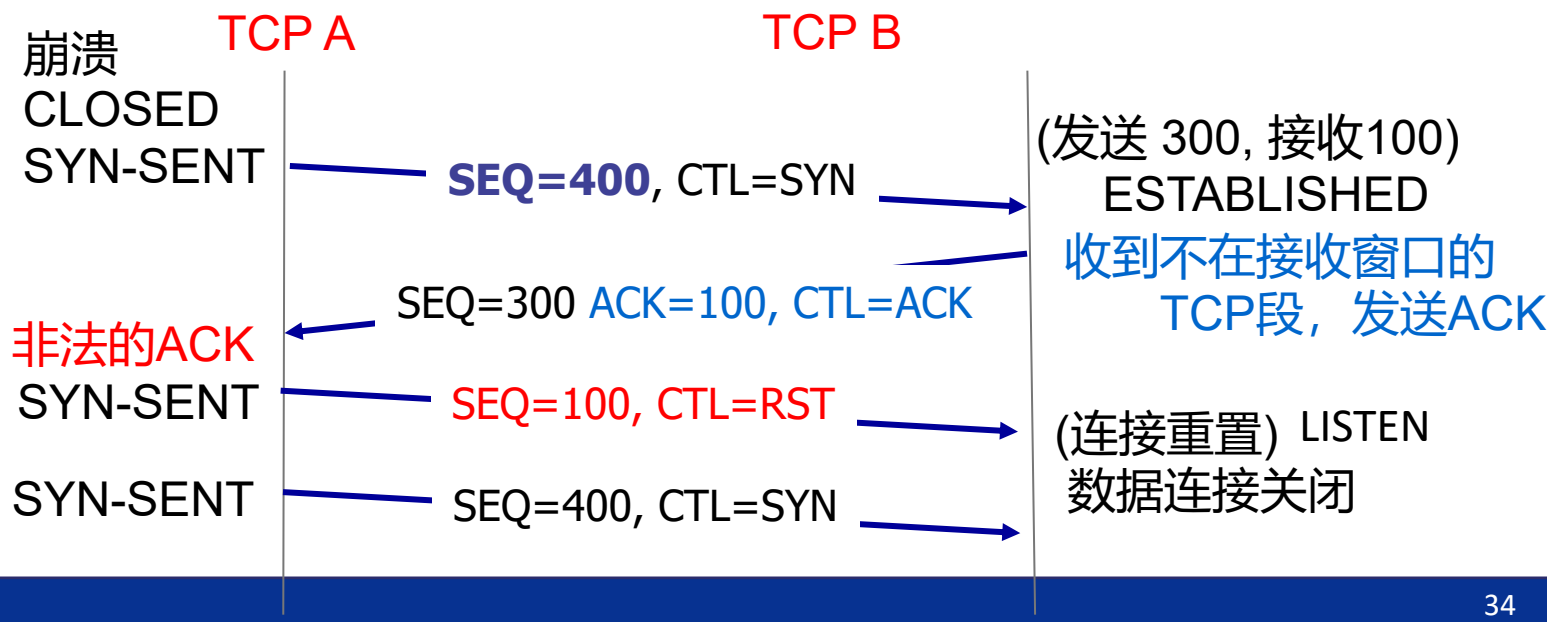
拓展的内容

- 连接已经建立后，可以发送RST=1的TCP段，重置连接，状态重置为CLOSED。数据可能丢失
- CLOSED状态，收到任何TCP段，都发送RST
- 在连接建立阶段（即非同步状态），收到一个明显不属于当前连接的TCP段时发送RST，以使通信双方重新同步
  - 某一方收到非法的数据段或非法的连接请求时发送ACK
  - 某一方收到非法的ACK(对方期待接收的并不在我的发送窗口中）时发送RST



考虑到分组可能丢失或失序到达，只要顺序号在接收窗口内，就是合法的RST段，导致可能的攻击：最坏情况下只需要发送 顺序号空间大小/接收窗口大小个RST

假设一个已经建立好的连接，接下来一方(TCP A)崩溃重启后，试图重新连接时正好对应着老的连接



TCP连接：重置(reset)连接

拓展的内容

- socket的**SO_LINGER**选项控制调用socket的**close()**方法时的行为
- SO_LINGER缺省为关闭状态，即发送FIN以优雅关闭连接
- 如果SO_LINGER选项打开：
  - 相应的linger time为0时，立刻发送RST终止连接
  - 相应的linger time不为0时，等待linger time(秒)，超时后发送RST

```
onoff = 1     # 是否打开SO_LINGER
linger_time = 0    # linger time多少

sock.setsockopt(socket.SOL_SOCKET, socket.SO_LINGER,
                struct.pack('ii', onoff, linger_time))
```

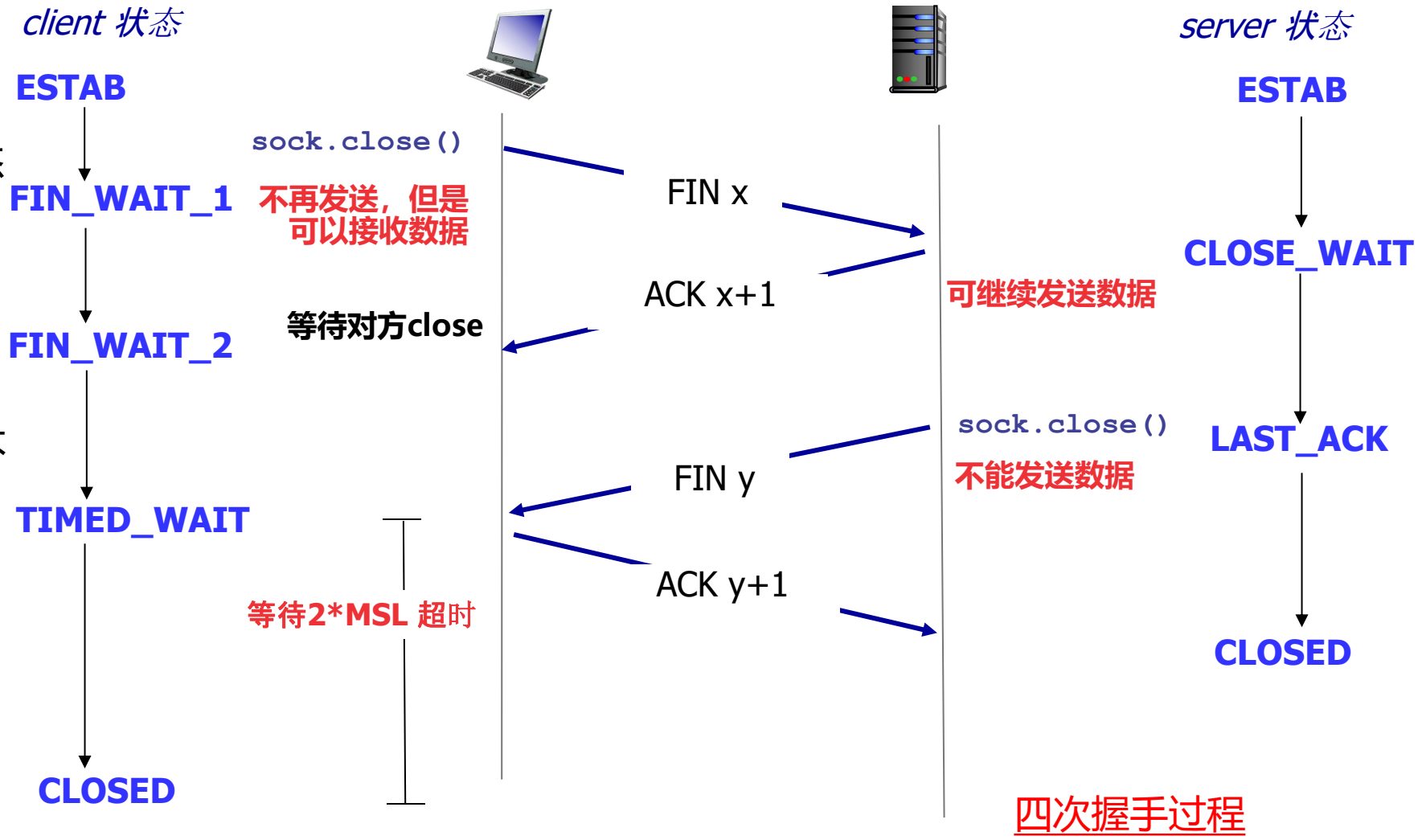
TCP连接：释放连接

连接释放采用四次（或三次）握手过程

- TCP连接可看成由两个方向的“单工”连接组成
- 两方都发送一个FIN标志置为1的TCP段，表示这个方向以后不再有数据要发送
- 另一方在收到FIN后，发送ACK来确认那个方向的连接被关闭
  - FIN可能丢失，需要对方发送ACK确认
  - 如果没有收到ACK时，FIN会进行重传
  - FIN占用一个顺序号，即一个方向的使用的顺序号空间为ISN, ISN+1,...FIN
  - FIN表示该方向不会再有正常的携带数据的TCP段，但是仍然可以接收另一方向发送的TCP数据，在收到数据后会发送ACK
  - 可以把对FIN的ACK与自身的FIN合并，相当于三次握手过程
- 只有当两个方向的连接均关闭后，该TCP连接才被完全释放
- 两个方向同时发送FIN也能够正常地释放连接

TCP连接释放： Client主动关闭连接

- 建议Client方主动关闭连接
- **主动关闭**的一端进入**TIME-WAIT状态**，需要维护连接状态2*MSL
  - 最后的ACK y+1可能丢失，另一方重传FIN y，收到后重传ACK y+1
  - 最坏需要等待2*MSL，确保不会有FIN Y到来，对新连接造成干扰
  - 新的连接建立请求到来时，可以检查对方使用的初始顺序号是否超过老连接上使用过的最大顺序号

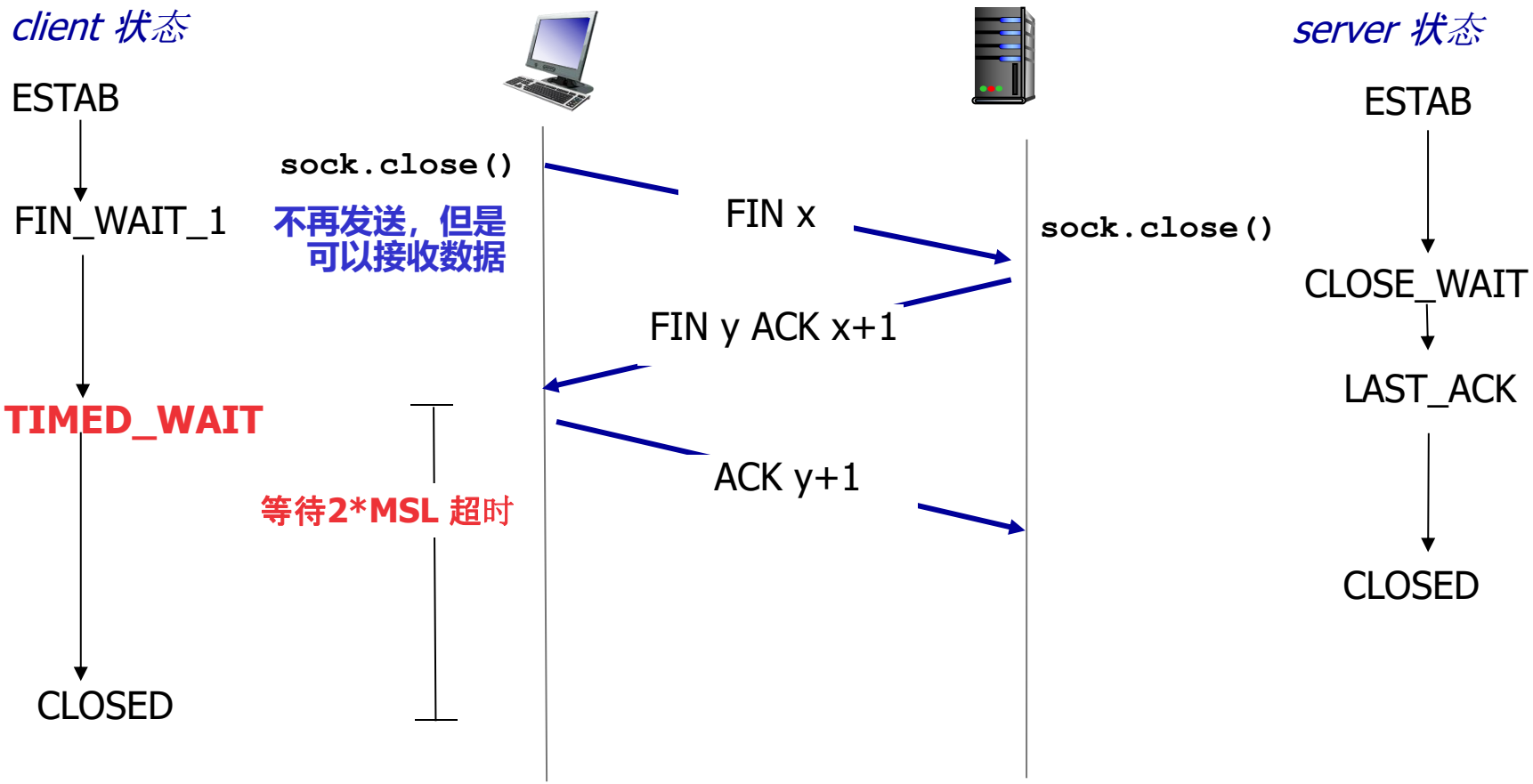


四次握手过程

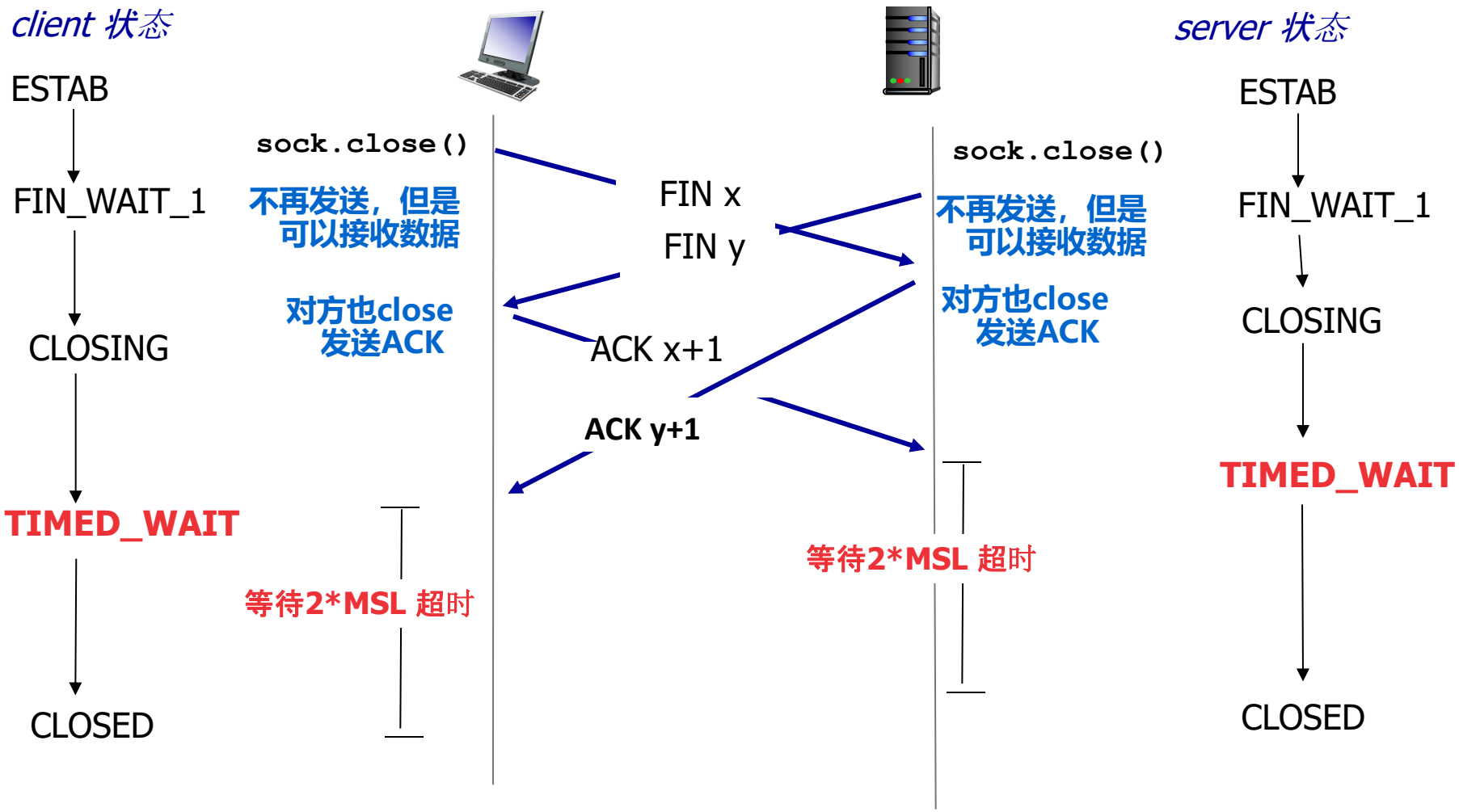
sock.shutdown(flag) flag可取SHUT_RD, **SHUT_WR**, SHUT_RDWR
sock.close() 缺省情况下，相当于 sock.shutdown(socket.SHUT_RDWR)

TCP连接释放： 三次握手过程

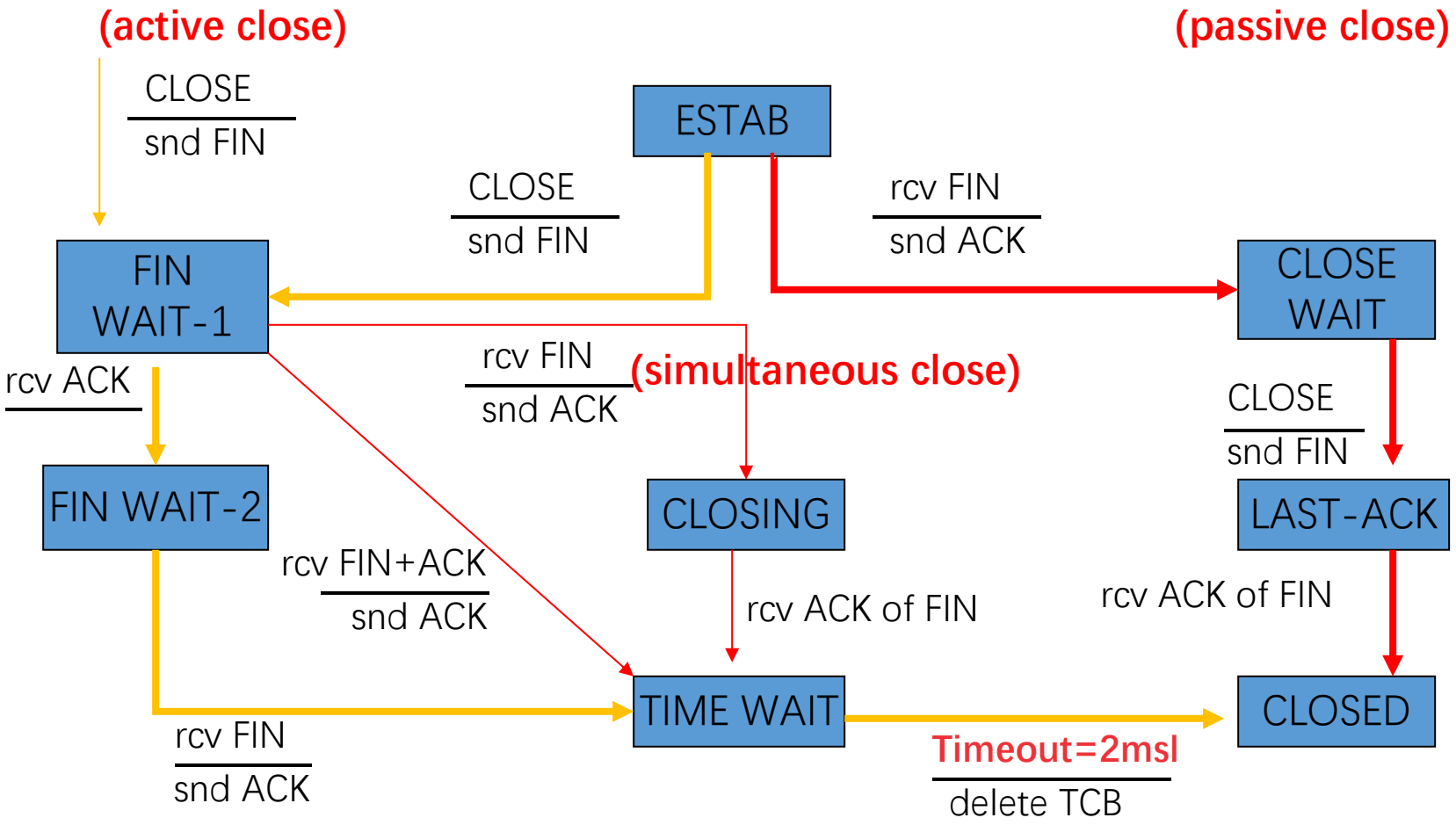
- 客户方主动关闭连接，服务方正好也要关闭连接，**捎带确认**，服务方的FIN和ACK合并在一起发送



TCP连接释放：双方同时主动关闭连接



TCP连接释放：状态转换图



教材目录

3.3 数据链路协议：可靠数据传输

- 停等协议
- 滑动窗口协议：GBN和选择重传

6.2.1 TCP概述

6.2.2 TCP段格式

6.2.4 TCP可靠传输

6.2.5 TCP流量控制

TCP可靠数据传输

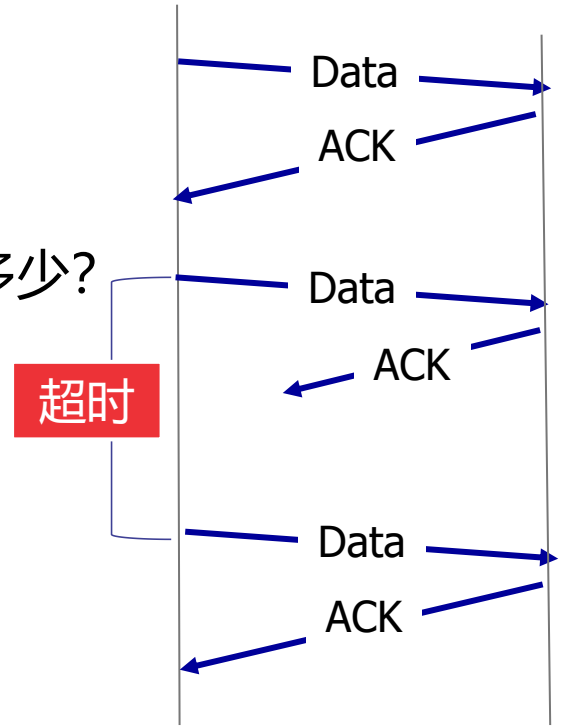
6.2.3 TCP连接管理：连接建立和连接释放

6.2.7 TCP计时器

6.2.6 拥塞控制

TCP数据段的重传超时计时器(RFC6298)

- 重传计时器超时：重传第一个尚未确认的TCP段
- 重传计时器的超时RTO(Retransmission Timeout)应该设置为多少？
  - 太长：低效（延时长，带宽利用率低）
  - 太短：造成不必要的重传（占用额外的带宽）
- TCP采用自适应的重传计时策略：基于RTT
  - EstimatedRTT存放估计的往返传输时间
  - SampleRTT为采样得到的当前RTT
    - 发送时记录时刻，收到该TCP段的ACK时记录时刻，两者的差值就是采样值
  - 采用指数加权平均EWMA(Exponential Weighted Moving Average)更新EstimatedRTT
    - 考虑最近多次采样
    - 之前的采样的权重成指数级下降，能够更快地对RTT采样的变化做出响应



采样点	权重%
1	12.5
2	10.94
3	9.57
4	8.37
5	7.33
6	5.61
7	4.91
8	4.30
9	3.76
10	3.29

$$weight_i = \alpha(1-\alpha)^{i-1}$$

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

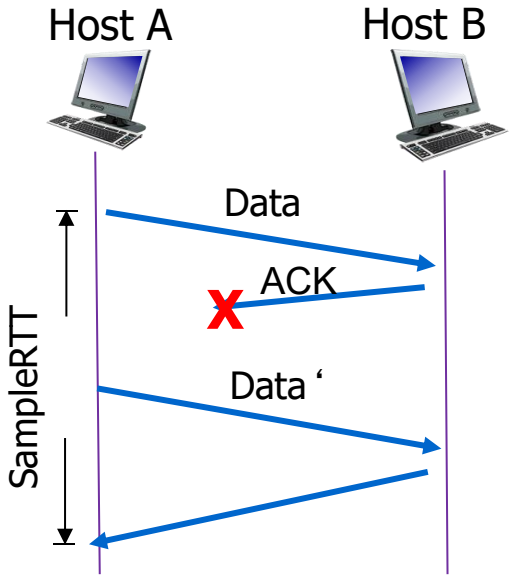
α=1/8

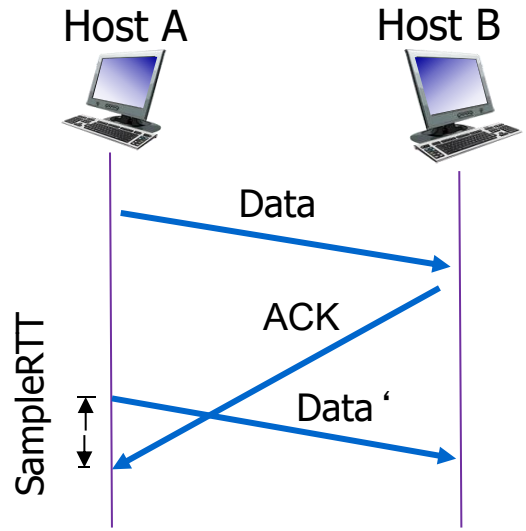
注意：我们这里采用RFC6298里面的计算公式，与教材采用的公式形式不同，但是完全等价

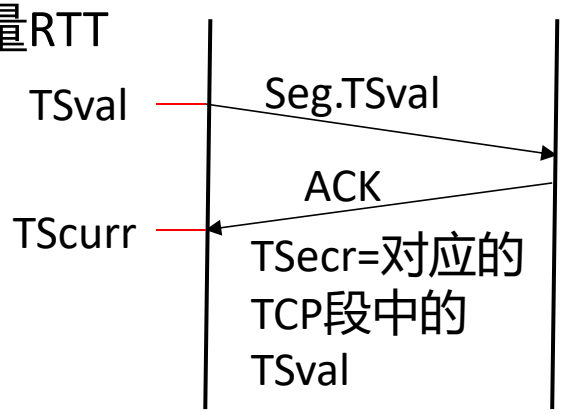
TCP数据段的重传超时计时器

如果TCP段超时重传时怎么办?

- 问题：对重传段的RTT测量有歧义
  - 收到的有可能是重传段的ACK
  - 收到的有可能是原TCP段的ACK
- Karn方法
  - 对重传的分组不测量RTT，不更新RTT估计
  - 重传分组的超时设置采用指数后退算法：原超时的2倍
- 拓展：TCP的时间戳选项TSOPT可以对重传段测量RTT







RTT = TScurr - TSval

TCP数据段的重传超时计时器

- 问题：RTT的浮动范围比较大，仅仅采用平均RTT来估算超时不是很精确
- Jacobson算法：考虑方差（安全边界)的因素。为了计算方便用平均绝对偏差（初始为SampleRTT/2）来代替均方差

$$Deviation = (1-\beta) \times Deviation + \beta \times |SampleRTT - EstimatedRTT|$$ β=1/4

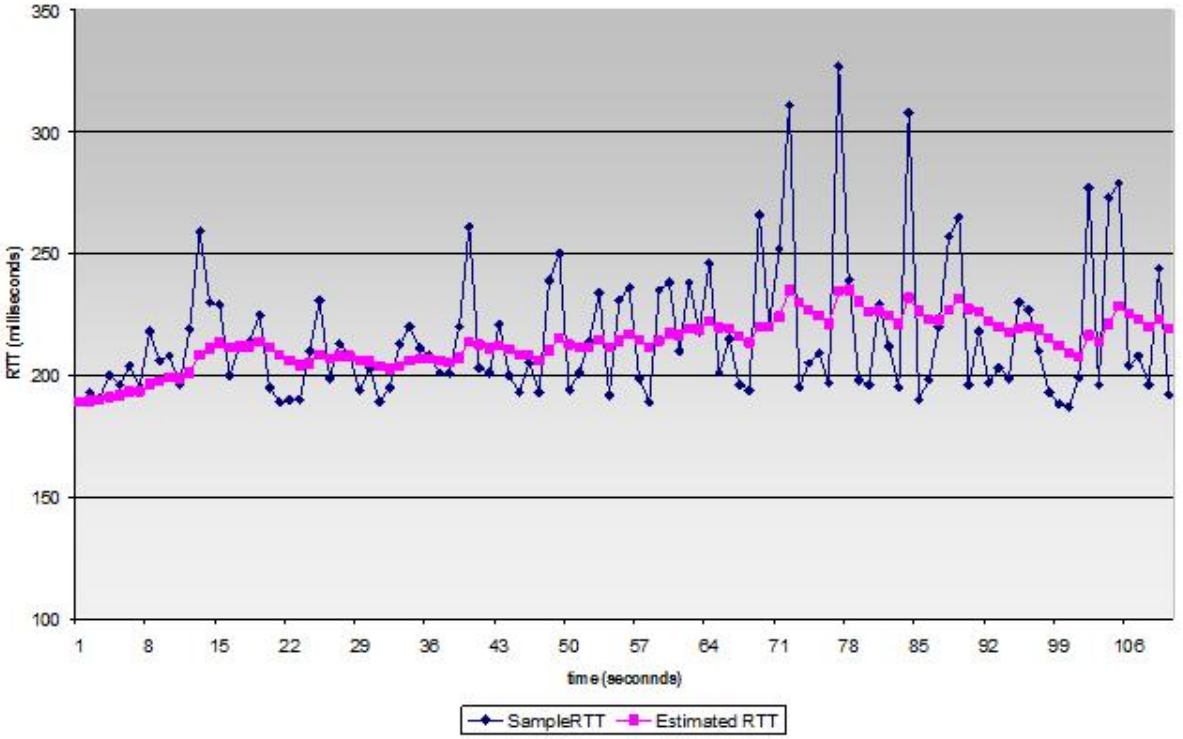
$$EstimatedRTT = (1-\alpha) \times EstimatedRTT + \alpha \times SampleRTT$$ α=1/8

第一次采样到来时：

$$EstimateRTT = SampleRTT$$
$$\text{Deviation} = \text{SampleRTT}/2$$

$$Timeout = EstimatedRTT + 4 \times Deviation$$

- 只对新的携带了用户数据的TCP段进行采样
- 重传TCP段的超时设置为上次超时的一倍



TCP的主要计时器

Timer	描述
Retransmission timer	重传TCP数据段
Retransmit-SYN timer	连接建立阶段使用，初始为1秒，指数后退，LInux实现中SYN重传最多6次(连接建立等待最多 2^6-1 = 63 秒)
TIME-WAIT计时器	主动关闭最后进入TIME-WAIT状态时设置，等待再次打开同一条连接的最小间隔时间，建议为2MSL (maximum segment lifetime)=240秒。Linux实现为60秒
Persist timer	收到接收窗口为0的ACK时设置，进行0窗口探测(ZWP，Zero Window Probe)避免数据传输中的死锁。初始为当前重传计时器的RTO，接受窗口仍然为0时，超时时间指数后退
Keepalive timer	检测连接是否活跃

TCP连接：保持连接活跃

- 如果A和B间的连接空闲一段时间，很有可能出现半开通(Half-Open)的连接
  - 某一方已经崩溃，丢失该连接的状态信息
  - 如果途中经过NAT设备，NAT设备可能不再维护那些不活跃的TCP连接
- TCP协议支持KeepAlive机制，以维持连接活跃
  - **连接空闲时开启KeepAlive计时器**（缺省7200秒=2小时），如果**有数据发送，停止该计时器**
  - KeepAlive计时器超时时：
    - **发送探测分组**，顺序号 = 收到的最大ACK号-1， 数据部分长度为0（缺省实现）；或者为了与早期的TCP实现兼容，发送长度为1，包含一个字节的垃圾数据（比如0）
      - 接收者发现顺序号在接收窗口之前会发送ACK作为回应
      - 发送者收到ACK时**重启KeepAlive计时器**
      - 如果没有回应，等待一段时间（75秒）重发探测分组
      - 多次探测（9次）后都没有响应时，连接终止
  - KeepAlive选项缺省关闭，这是考虑到有可能暂时的通信故障，反而导致连接被不正常终止

```
sock.setsockopt(socket.SOL_SOCKET, socket.SO_KEEPALIVE, True)
```