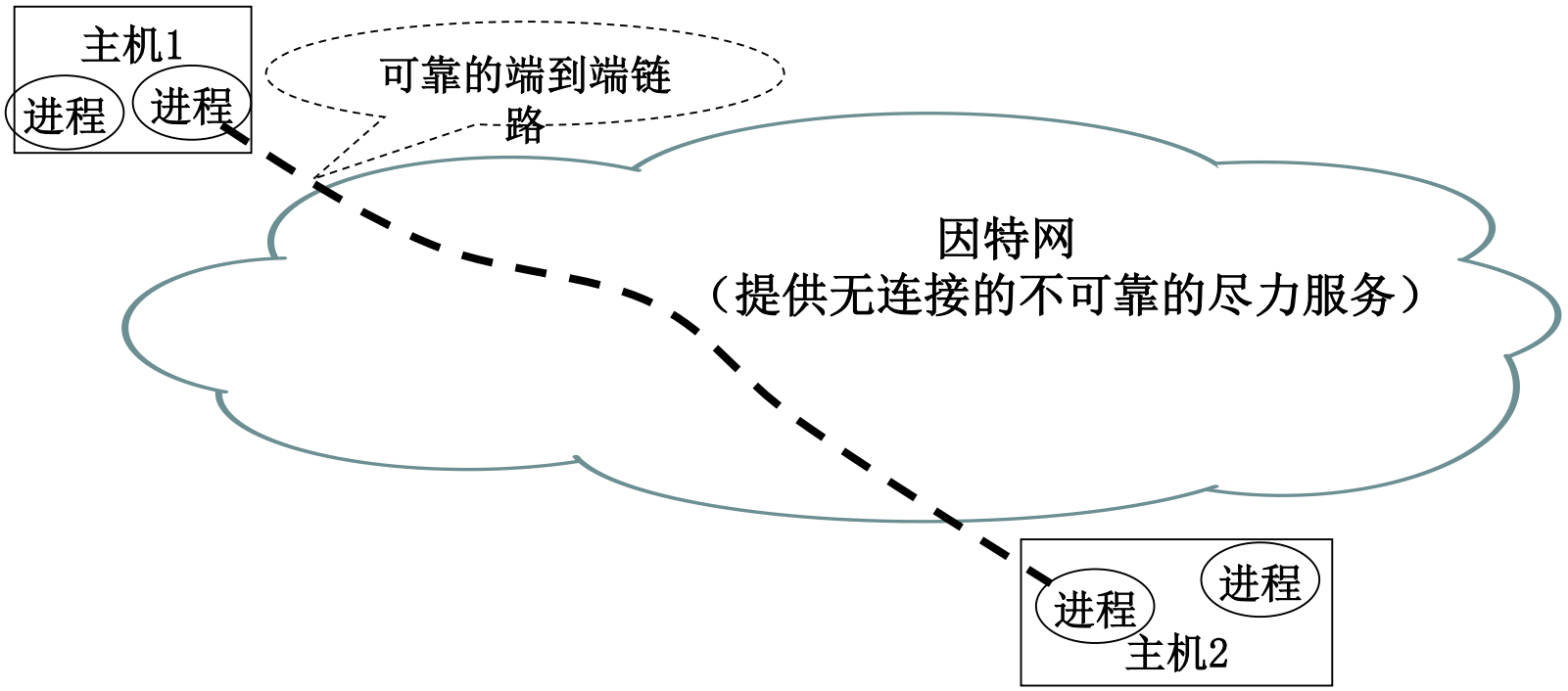
第六单元 传输层

2019.5.29

本节内容

- 概述
- TCP/UDP报文
- 传输层的多路复用
- 端口号
- UDP协议
- 传输控制协议(TCP)
- TCP报文格式
- 三次握手建立连接
- 四次握手释放连接
- TCP状态转换图
- TCP滑动窗口
- 快速重传
- 延迟确认
- 选择性确认
- TCP超时计算
- 拥塞控制
- 问题1：长肥管道
- 问题2：死锁现象
- 问题3：傻瓜窗口症候
- TCP定时器

概述



- 传输层协议称为**端到端或进程到进程的协议**。因特网的传输层可以为两个进程在不可靠的网络层上建立一条**可靠的逻辑链路**，可以提供**字节流**传输服务，并且可以进行**流控制**和**拥塞控制**。

TCP/UDP报文

- **因特网的传输层**有两个协议：**UDP**和**TCP**。UDP协议提供不可靠的尽力服务，TCP协议提供可靠的字节流服务。

IP头部	UDP报文/TCP报文

协议号：TCP=6　　ICMP=1
　　UDP=17　　IGMP=2

- 我们把传输层的数据单元（报文）称为数据段(segment)。

TCP: http://tools.ietf.org/html/rfc793
　　http://en.wikipedia.org/wiki/Transmission_Control_Protocol
UDP: http://tools.ietf.org/html/rfc768
　　http://en.wikipedia.org/wiki/User_Datagram_Protocol

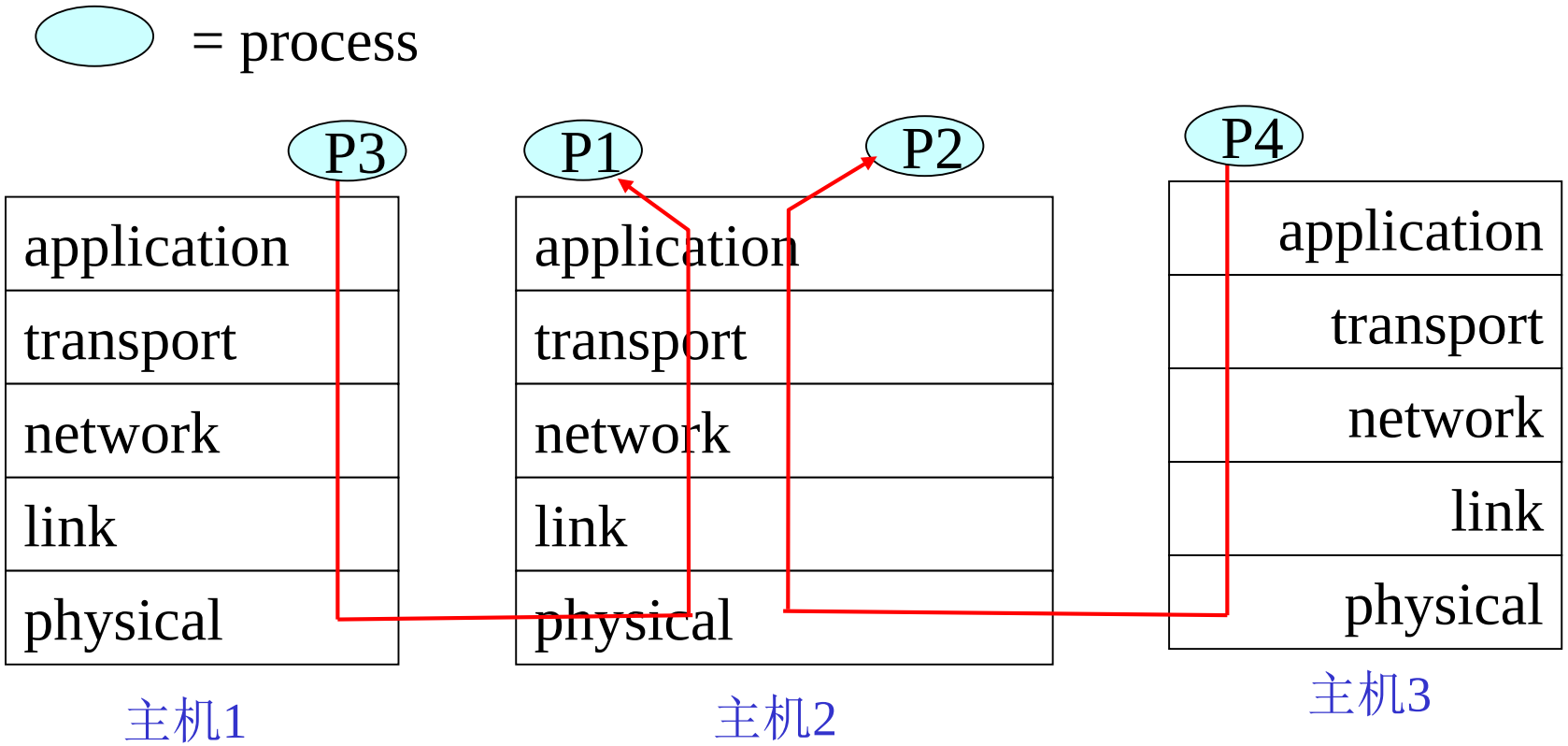
传输层的多路复用和解多路复用

接收方解多路复用:

把收到的数据段交给正确的上层进程

发送方多路复用:

收集来自上层进程的数据形成数据段并发送出去



端口号

UDP协议和TCP协议如何知道把收到的数据段交给哪个上层进程呢？利用数据段(segment)中的目的端口号 (2个字节)

- 知名端口

  0~1023。为提供知名网络服务的系统进程所用。
  例如: 80-HTTP，21-ftp Control，20-ftp Data，23-telnet，
  25-SMTP，110-POP3，53-DNS

- 注册端口

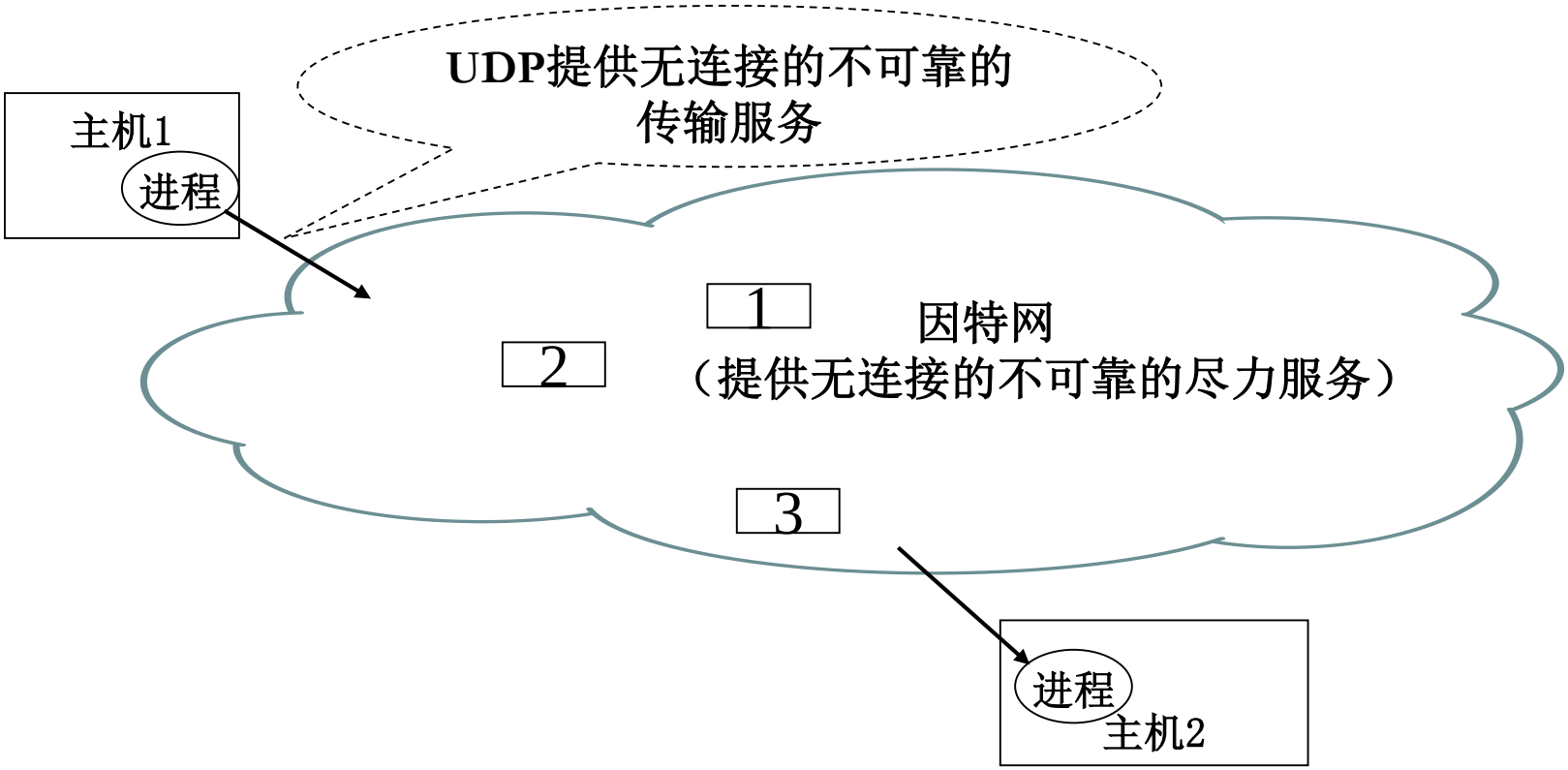
  1024~49151。在IANA注册的专用端口号，为企业软件所用。

- 动态端口

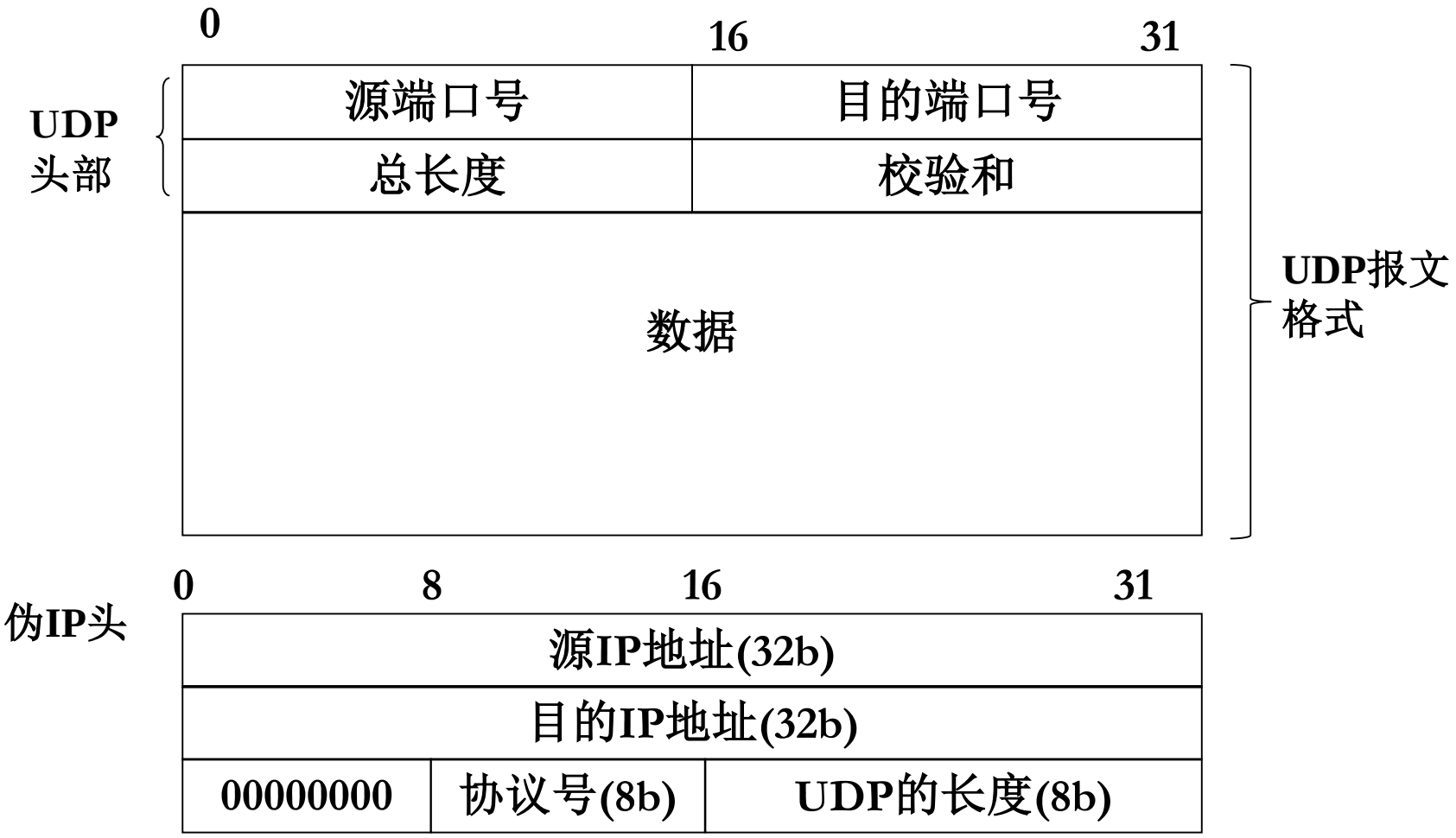
  49152~65535。没有规定用途的端口号，一般用户可以随意使用。也称为私用或暂用端口号。

http://en.wikipedia.org/wiki/Well_known_ports#Well-known_ports

UDP协议



- 用户数据报协议(User Datagram Protocol)只提供无连接的不可靠的尽力服务。发送给接收进程的数据有可能丢失，也有可能错序。
- 接收进程每次接收一个完整的数据报，如果进程设置的接收缓冲区不够大，收到的数据报将被截断。

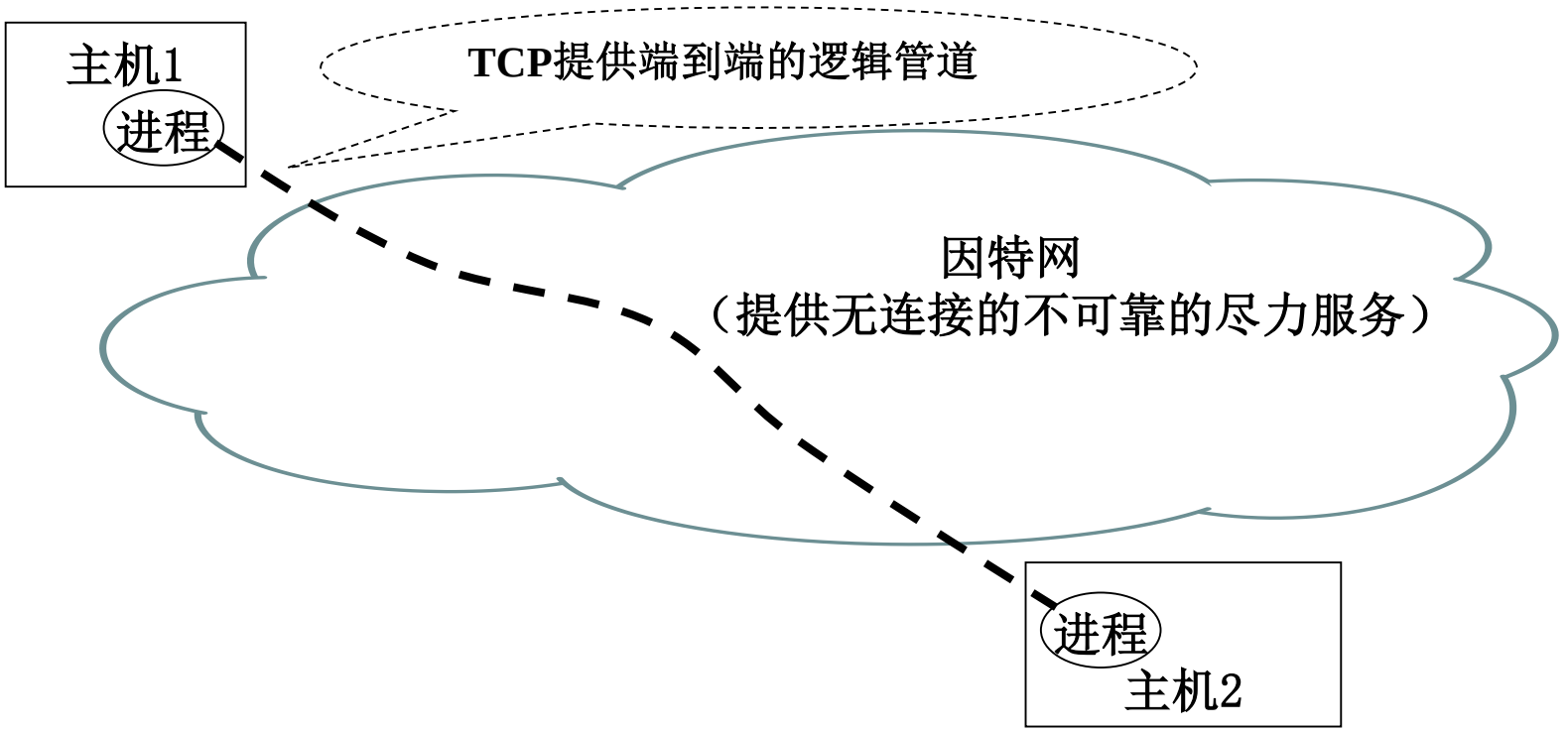


总长度： 整个UDP报文的长度。

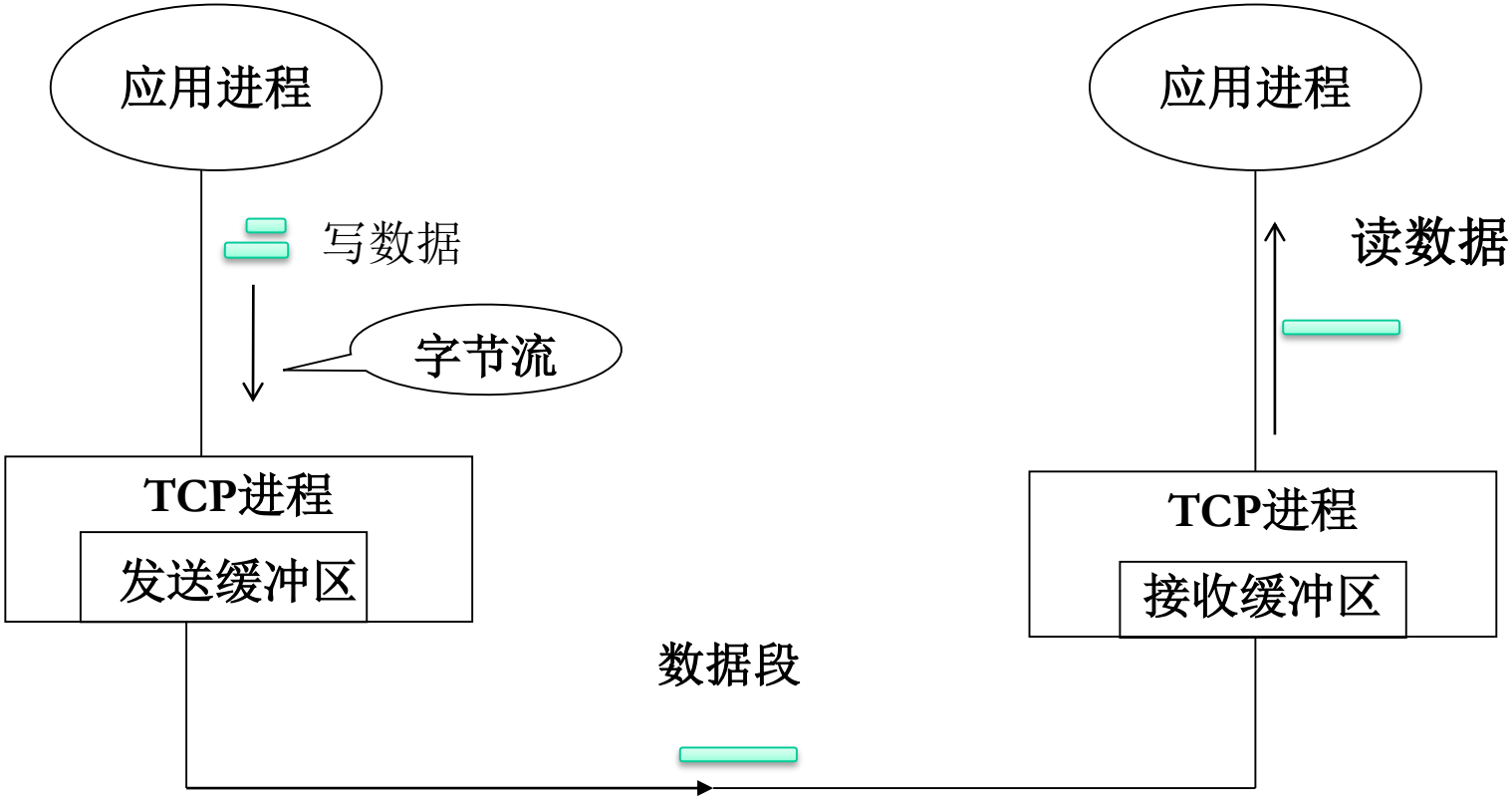
源端口号和目的端口号： 用于关联发送进程和接收进程。

校验和： 由伪IP头、UDP头(校验和为0)和UDP数据形成。其中，伪IP头的协议号为17。如果发送方把校验和设置为0，接收方会忽略校验和 。UDP长度就是UDP头部的总长度。

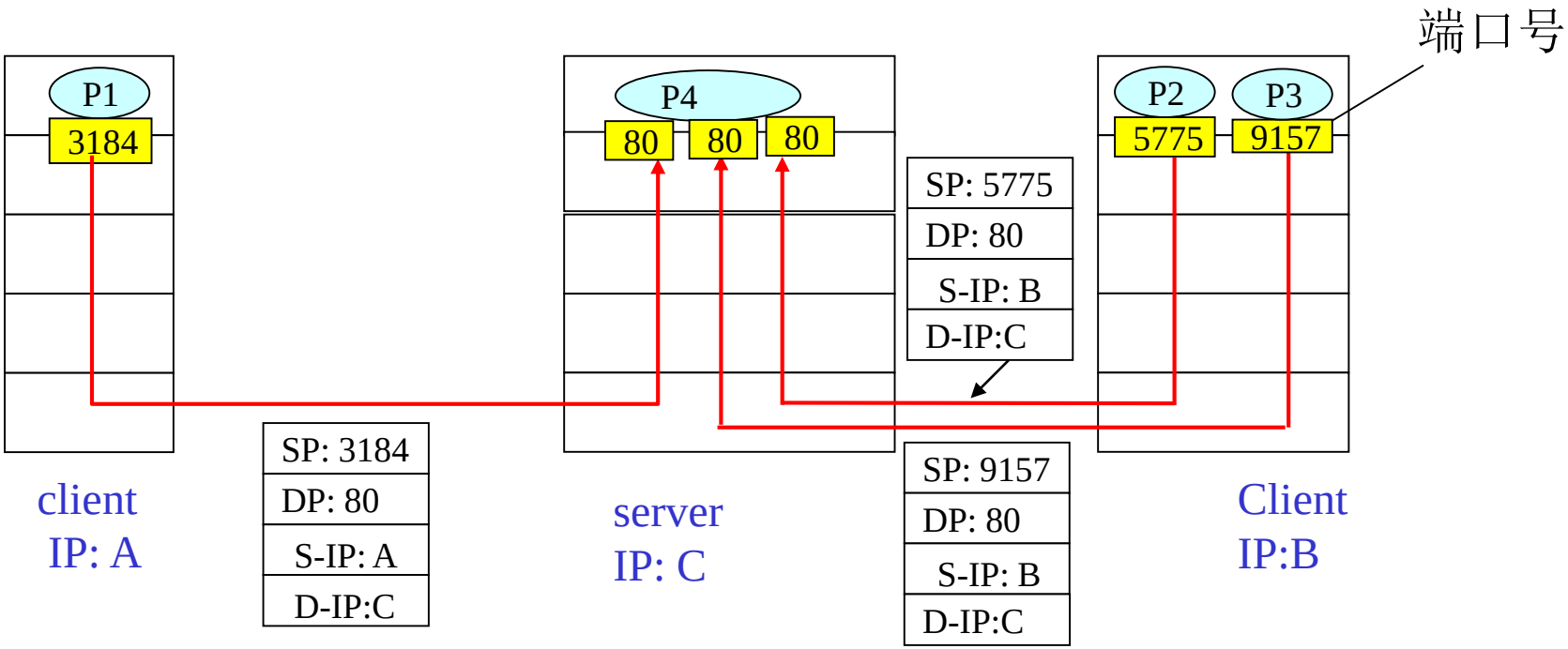
传输控制协议(TCP)



- TCP(Transmission Control Protocol)为进程之间提供面向连接的**可靠的数据传送服务**。
- TCP为**全双工**协议。TCP提供**流控制**机制，即控制发送方的发送速度，使发送的数据不会淹没接收方。作为因特网的主要数据发源地，TCP还提供**拥塞控制**功能。

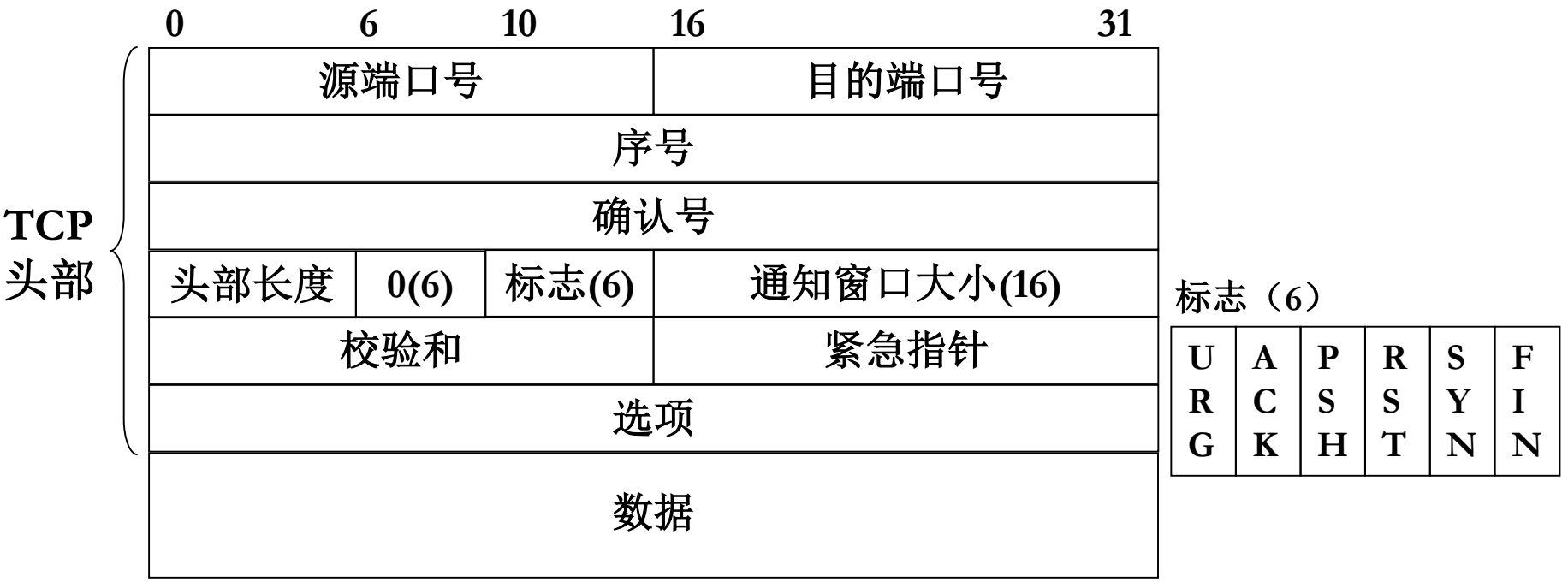


- 一个TCP连接提供可靠的字节流服务。字节流服务表示没有消息边界。例如，多次发送的数据可以放在一个数据段中传送且不标识边界。
- 每个数据段的数据部分的最大长度(字节)不能超过MSS (Maximum Segment Size) 。
- 每个TCP连接可以由四元组唯一标识：源IP地址, 源端口号,目的IP地址,目的端口号。

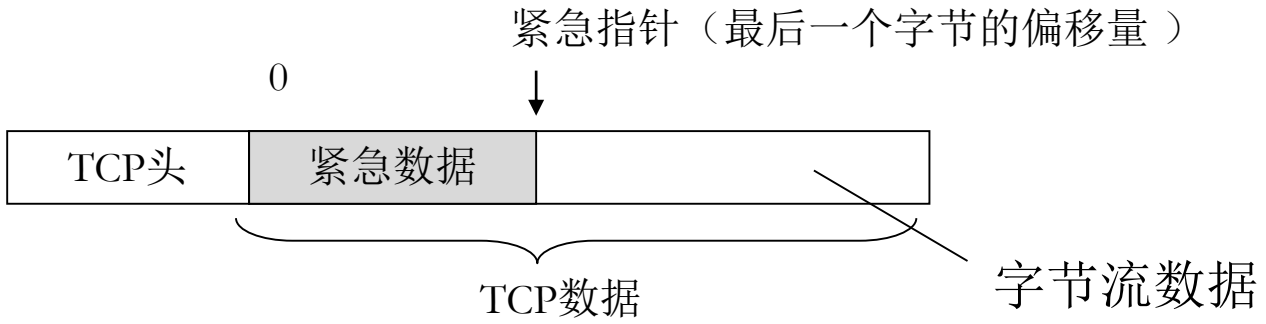


SP: source port#
DP: dest port#
S-IP: source IP addr
D-IP: dest IP addr

TCP报文格式


0
6
10
16
31
TCP
头部
源端口号
目的端口号
序号
确认号
头部长度
0(6)
标志(6)
通知窗口大小(16)
校验和
紧急指针
选项
数据
标志（6）
URG
ACK
PSH
RST
SYN
FIN

- 字节流中的每个字节均被编号。初始序号采用基于时间的方案， 一般采用随机数。数据部分的第一个字节的编号为初始序号加1。
- 每个数据段的**序号**采用其数据部分第一个字节的编号。**确认号**为期待接收的下一个数据段的序号。只有设置了确认标志，确认号才有效。
- **头部长度**以四个字节为单位。
- **校验和**由伪IP头、TCP头和TCP数据部分形成。其形成方法与UDP协议类似。
- **紧急指针**用于指出带外数据(out-of-band)的边界。标志URG为1时有效。



- 发送窗口为确认号之后发送方还可以发送的字节的序号范围。接收方用**通知窗口大小(advertised window)**告知发送方接收窗口的大小，发送方会据此修改发送窗口大小。
- 建立连接时的**选项**： MSS(Maximumu Segment Size)、窗口比例(Scale)；是否使用选择性确认(SACK-Permitted)。 Unix系统的默认值：MSS为536，SACK-Permitted为False。Windows 的默认值MSS为1460，SACK-Permitted为True。
- 数据传送时的选项：选择性确认的序号范围(Seletive ACK,SACK)，时间戳等。

URG: 表示本数据段包含紧急数据。

ACK: 表示确认号有效。

PSH: 告知接收方发送方执行了推送(Push)操作，接收方需要尽快将所有缓存的数据交给接收进程。

RST: 重置(Reset)连接。因为连接出现了错误，通知对方立即中止连接并释放相关资源。

SYN: 同步(Synchronize)序号标志，用来发起一个TCP连接。

FIN: 结束(Finish)标志，向对方表示自己不再发送数据。

* 所有标志为1有效。

TCP协议的工作过程

非对称活动

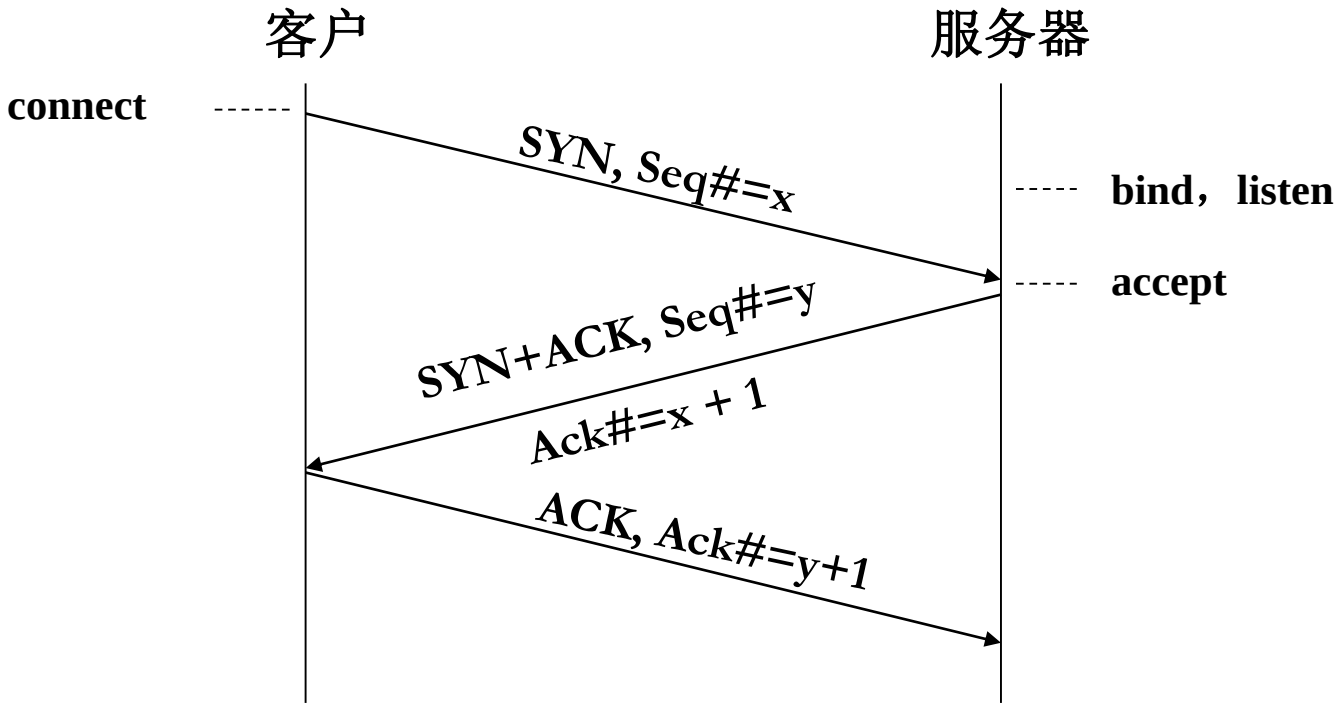
客户向服务器呼叫

全双工

对称活动

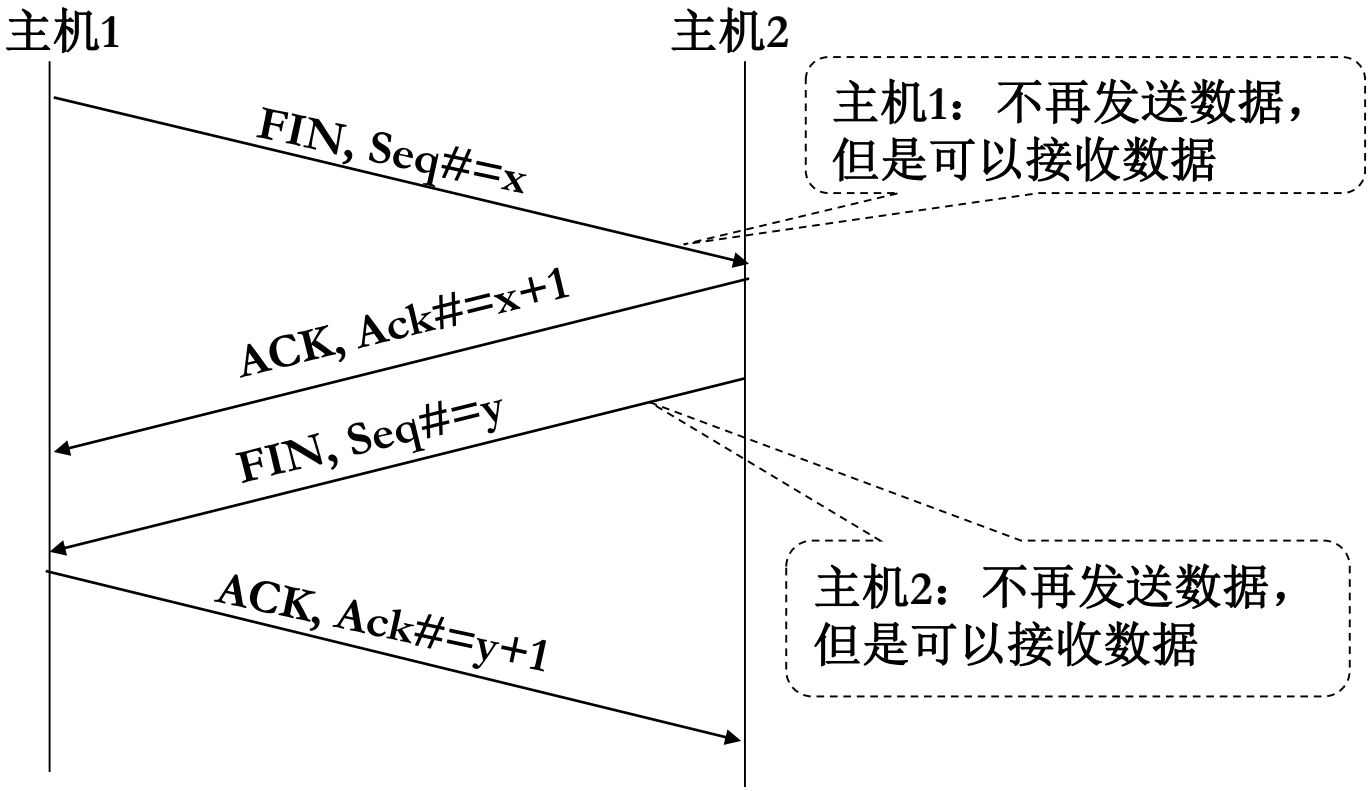
可由任何一方发起

三次握手建立连接

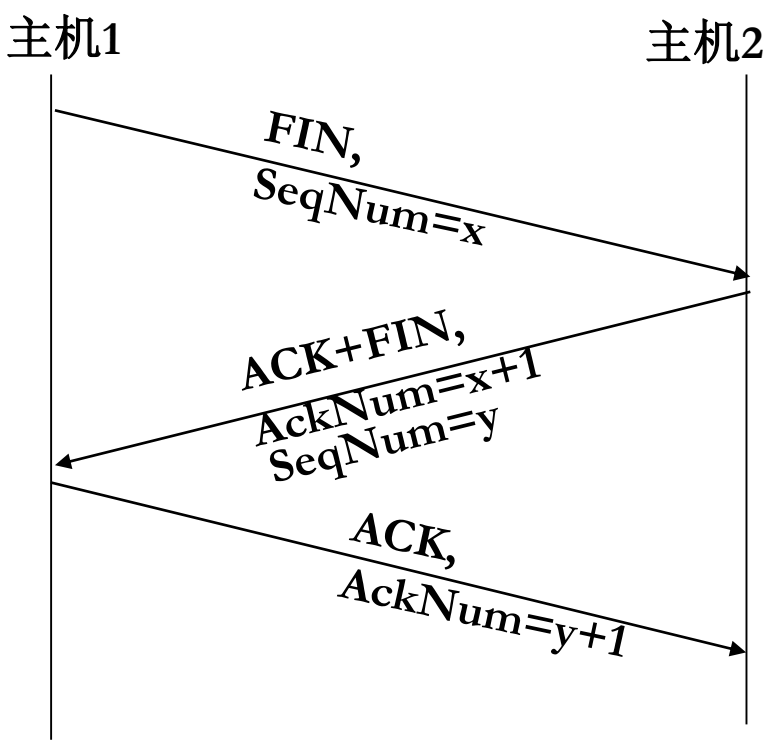


- 客户端需要知道服务器的IP地址和端口号。服务器收到客户端发来的连接请求(SYN报文)后查看是否有进程监听该端口，若有，则将此连接请求传给该进程，否则，服务器发RST拒绝它。如果该进程接受连接请求，则发回SYN+ACK报文，。
- 每一步均采用超时重传，多次重发后将放弃。重发次数与间隔时间依系统不同而不同。
- x和y为初始序号，分别用于两个方向的数据传送，它们一般采用基于时间的方案，如：随机数。两个方向的下一个数据段的序号分别为x+1和y+1。
- 头两个数据段确定的选项： Scale, MSS（告诉对方期待接收的值),SACK-Permited

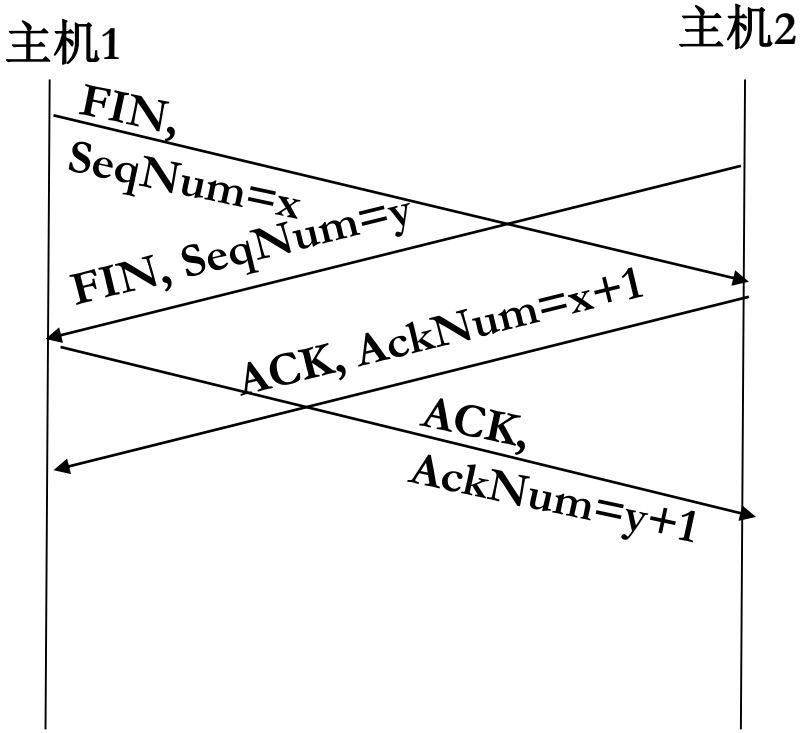
四次握手关闭连接



- FIN报文采用超时自动重发方式。在若干次重发后依然没有收到确认，则发送RST报文给对方后强行关闭连接。不同的系统重发方法不同。x和y都是上一个收到的数据段的确认号。
- 先发送FIN报文的一方在ACK发送完毕后需要等待2MSL(Maximum Segment Lifetime)的时间才完全关闭连接。TCP标准中MSL采用60秒，Unix采用30秒。

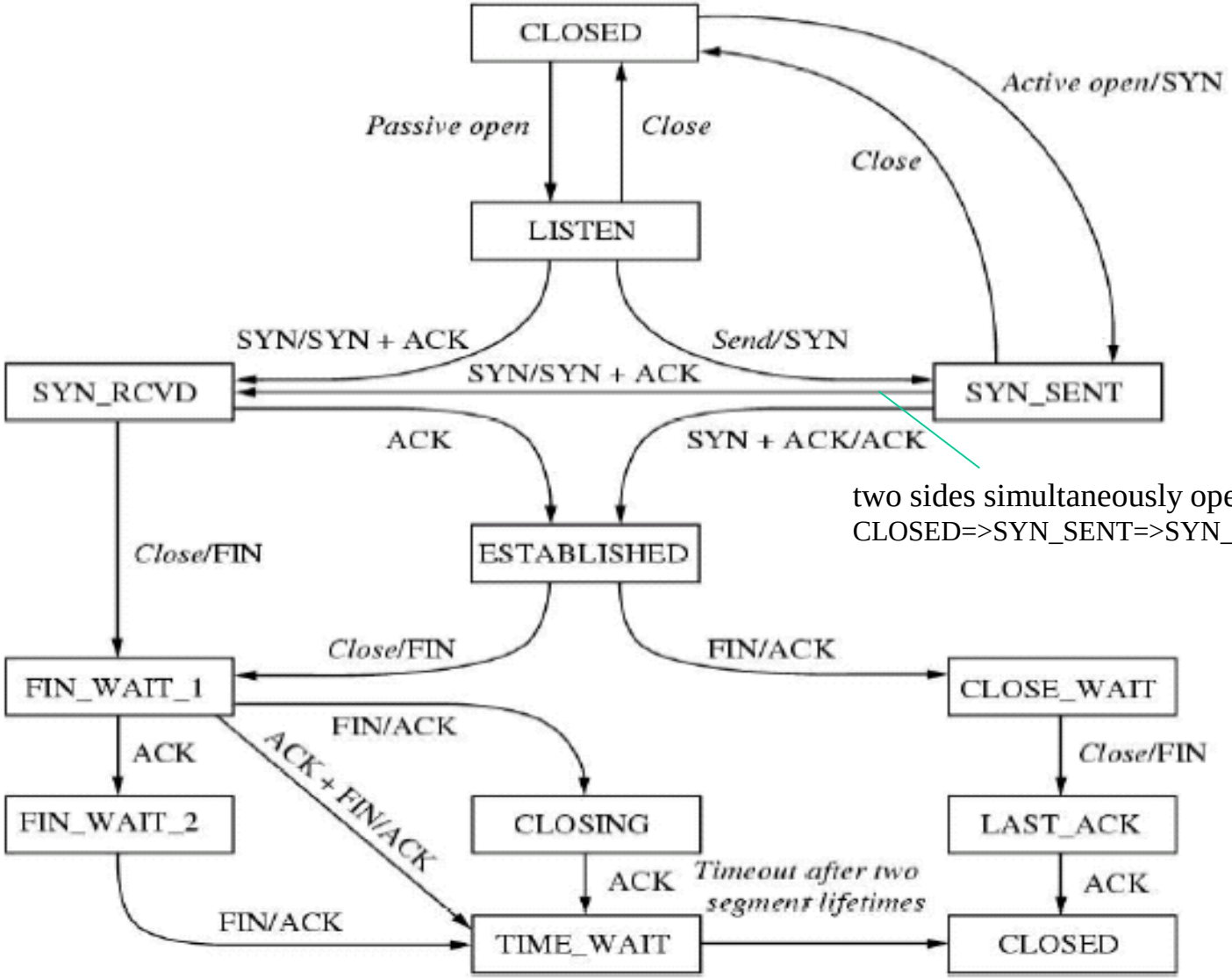


合并中间两次握手(ACK和FIN)



双方同时发出FIN

TCP状态转换图



two sides simultaneously open:
CLOSED=>SYN_SENT=>SYN_RCVD=>ESTABLISHED

建立连接：

（1）主动连接：CLOSED → SYN_SENT → ESTABLISHED

（2）被动连接：CLOSED → SYN_RCVD → ESTABLISHED

（3）同时打开：CLOSED → SYN_SENT → SYN_RCVD → ESTABLISHED

释放连接：

（1）一方先关闭： ESTABLISHED → FIN_WAIT1 → FIN_WAIT2
→ TIME_WAIT→ CLOSED

（2）另一方先关闭：ESTABLISHED → CLOSE_WAIT → LAST_ACK → CLOSE

（3）双方同时关闭：ESTABLISHED → FIN_WAIT1 → CLOSING → TIME_WAIT
ESTABLISHED → FIN_WAIT1 → TIME_WAIT

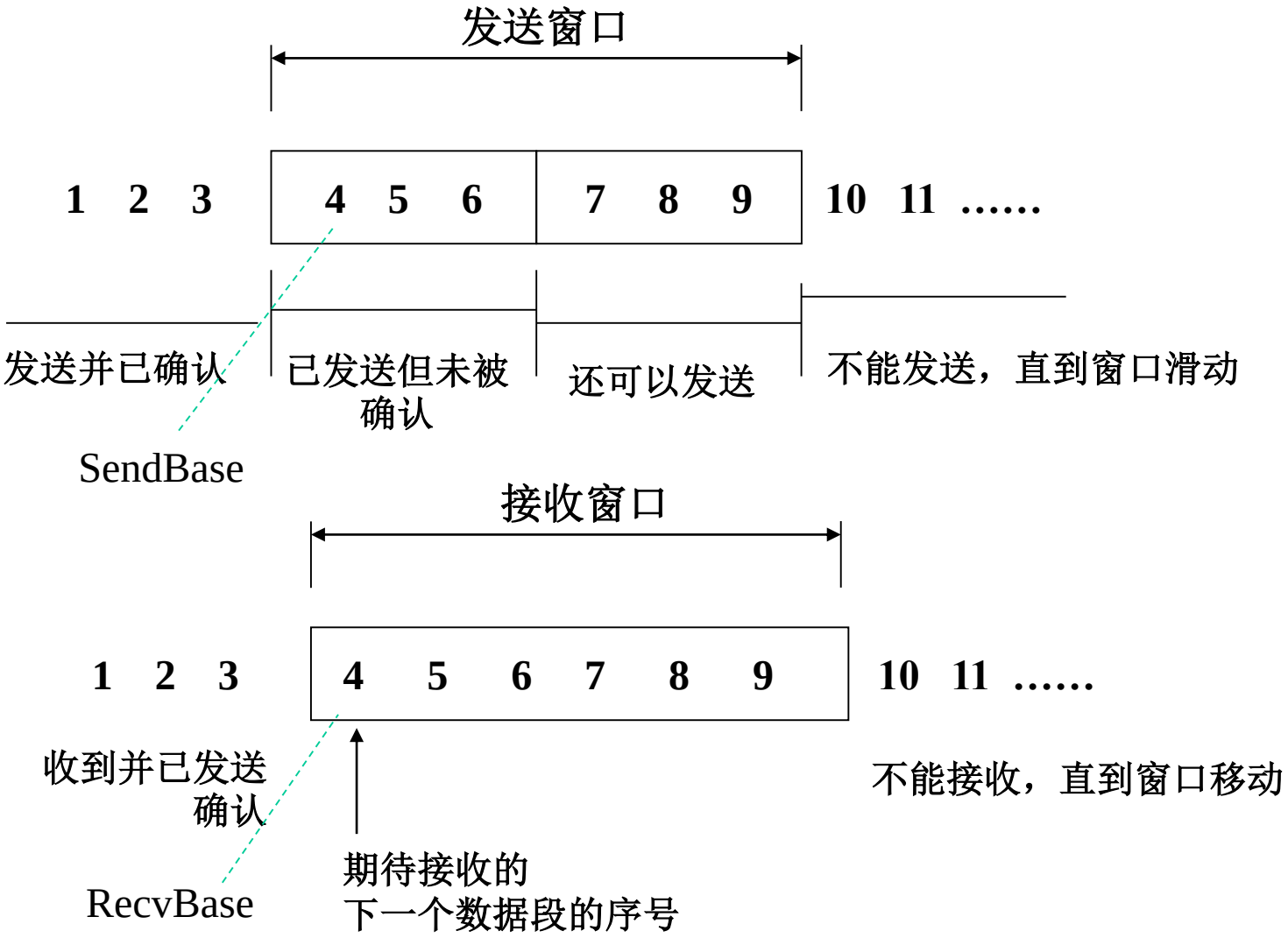
TCP滑动窗口

- TCP协议使用选择性确认，不使用NAK。
- 只有一个超时定时器。
- 采用字节流方式，每个数据段使用其第一个字节的编号作为**序号**。**确认号**为期待接收的下一个字节(下一个数据段)的序号。

  - TCP连接: MSS=1000 x(初始序号)=1999 发送的字节数=2500

Seq#	2000	3000	3500
	1000	500	1000

- TCP协议没有说明如何处理错序到达的数据段，要取决于具体实现。



* advWin为接收窗口的大小，即空闲块的大小（包含错序到达的数据段）

TCP滑动窗口—发送方

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
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)
    event: timer timeout
      retransmit not-yet-acknowledged segment with
            smallest sequence number
      restart timer
    event: ACK received from IP, with ACK field value of y
      if (y > SendBase) {
          SendBase = y
          if (there are currently not-yet-acknowledged segments)
                start timer
      }
} /* end of loop forever */
```

只有一个超时定时器

Comment:

- SendBase-1: last cumulatively ack'ed byte

Example:

- SendBase-1 = 71; y= 73, so the rcvr wants 73+ ; y > SendBase, so that new data is acked

TCP滑动窗口—接收方

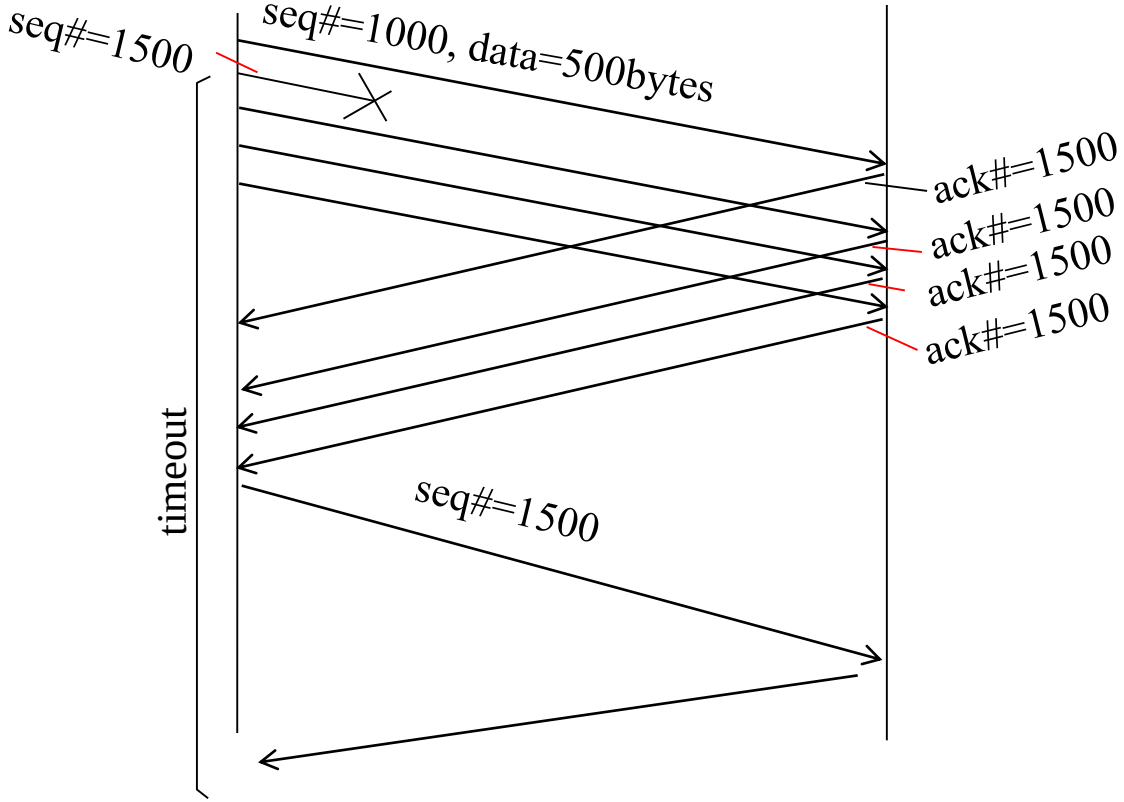
```
NextSeqNum = InitialSeqNum
RECVBase = InitialSeqNum
loop (forever) {
   switch(event)
     event: DATA fetched from application above
         send ACK
               with ACK.win = size of free buffer block
                     ACK.ack#= NextSeqNum
     event: DATA received from IP
        y = DATA.SEQ# field
        if (y == RecvBase) {                    （没考虑错序到达的数据段）
            RecvBase = y + DATA.length
        }
        send ACK
              with ACK.win = size of free buffer block
                     ACK.ack#= RecvBase
} /* end of loop forever */
```

cumulative acknowledge: only acknowledge to the expected segment.

累积确认

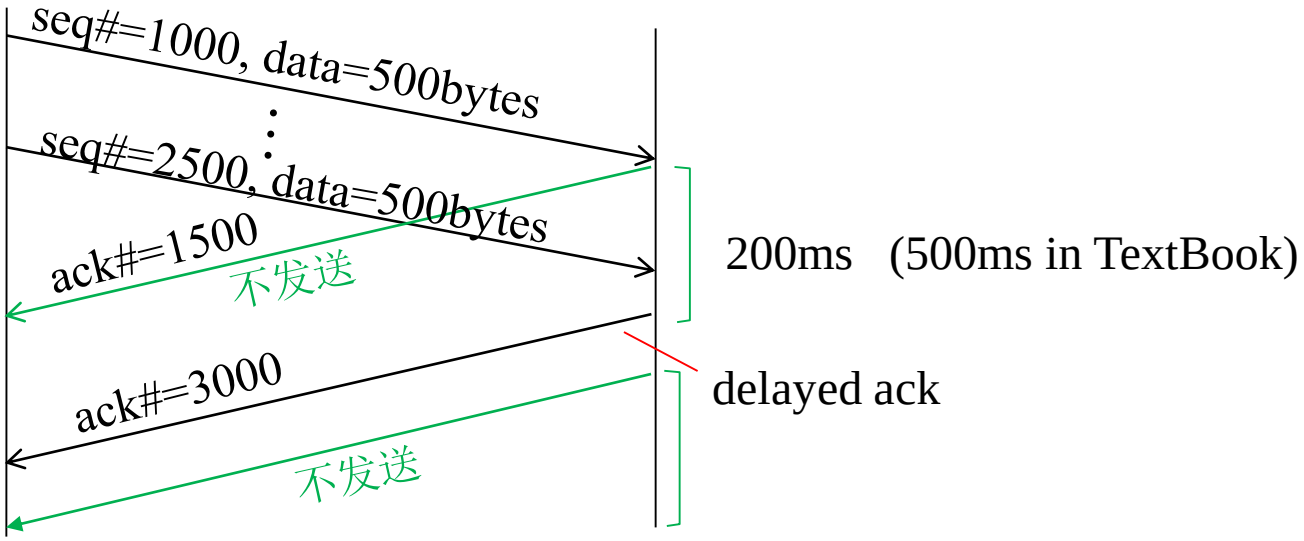
快速重传

- 如果发送方收到一个数据段的3次重复的ACK，它就认为其后的数据段（由确认号指出）已经丢失，在超时之前会重传该数据段，这种方法称为快速重传(fast retransmit)。



延迟确认

- 采用**延迟确认(*delayed ACK*)**时，接收方并不在收到数据段立即进行确认，而是延迟一段时间再确认。如果这个期间收到多个数据段，则只需要发送一个确认。如果在这个期间接收方有数据帧要发往发送方，还可以使用**捎带确认(*piggybacking*)**。
- 大部分的系统 (Windows, Unix)的延迟确认时间为200毫秒。TCP标准要求延迟确认不大于500毫秒。



选择性确认

选择性确认允许接收方把收到的数据块通过数据段的选项告知发送方，使发送方不会重传这些数据块。

received segment

Triggering Segment	ACK	First Block Left Edge	First Block Right Edge	2nd Block Left Edge	2nd Block Right Edge	3rd Block Left Edge	3rd Block Right Edge
5000	5500						
5500	(lost)						
6000	5500	6000	6500				
6500	(lost)						
7000	5500	7000	7500	6000	6500		
7500	(lost)						
8000	5500	8000	8500	7000	7500	6000	6500
8500	(lost)						

1st 2nd 3th

TCP SACK Option

=26

Kind=5	Length
Left Edge of 1st Block	
Right Edge of 1st Block	
Left Edge of 2th Block	
Right Edge of 2th Block	
Left Edge of 3th Block	
Right Edge of 3th Block	

http://tools.ietf.org/html/rfc2018

http://tools.ietf.org/html/rfc2883

TCP超时计算- 初始公式

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

- $0<\alpha\leq1$, α越小过去样本的影响越大。
- 一般取值: α =0.9。这会使过去影响指数减少。
- 这个公式也称为指数加权移动平均方法 (Exponentially Weighted Moving Average).
- RTO(retransmission timeout) = 2x EstimatedRTT

这个公式的另一种表示方法：

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

* 蓝色表示变量的旧值

TCP超时计算—Jacobson算法

- 修改上一页公式的参数：α =1/8。
- **Jacobson/Karels提议RTO**计算还要加上一个合适的安全边际(safety margin)，使得在样本变化较大时RTO会很快变得更大。

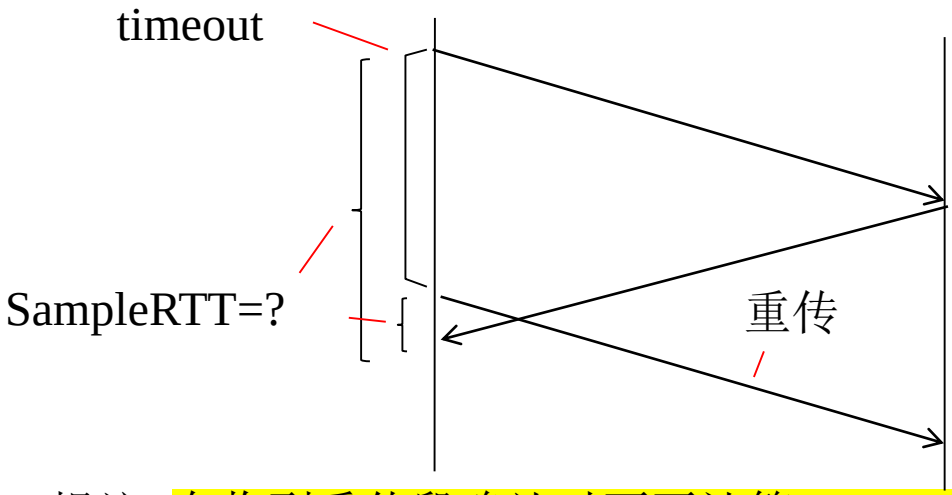
DevRTT =(1-β)*DevRTT + β*|SampleRTT-EstimatedRTT|
(typically, β = 1/4)

RTO = EstimatedRTT + 4*DevRTT

RFC1122，RFC 2988

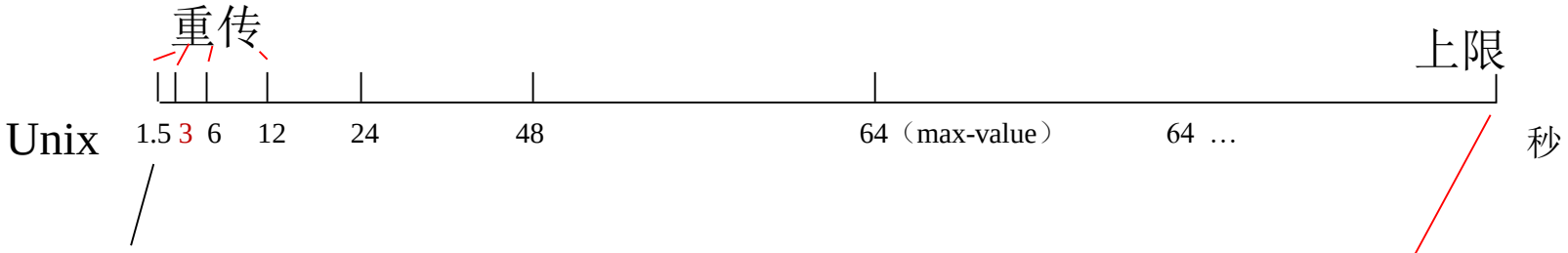
采样频率：如果发送窗口为12MSS，则每12个段取样一次。

TCP超时计算—Karn算法



RTO = γ × RTO (γ=2)

Phil Karn提议: 在收到重传段确认时不要计算EstimatedRTT。在每次重传时直接把RTO加倍直到数据段首次得到确认，并把这个RTO作为后续段的RTO。这个修正称为**Karn算法**(Karn and Partridge,1987)。 每次收到非重传段的ACK之后，就计算EstimatedRTT和正常的RTO，并用这个RTO作为后续段的RTO 。



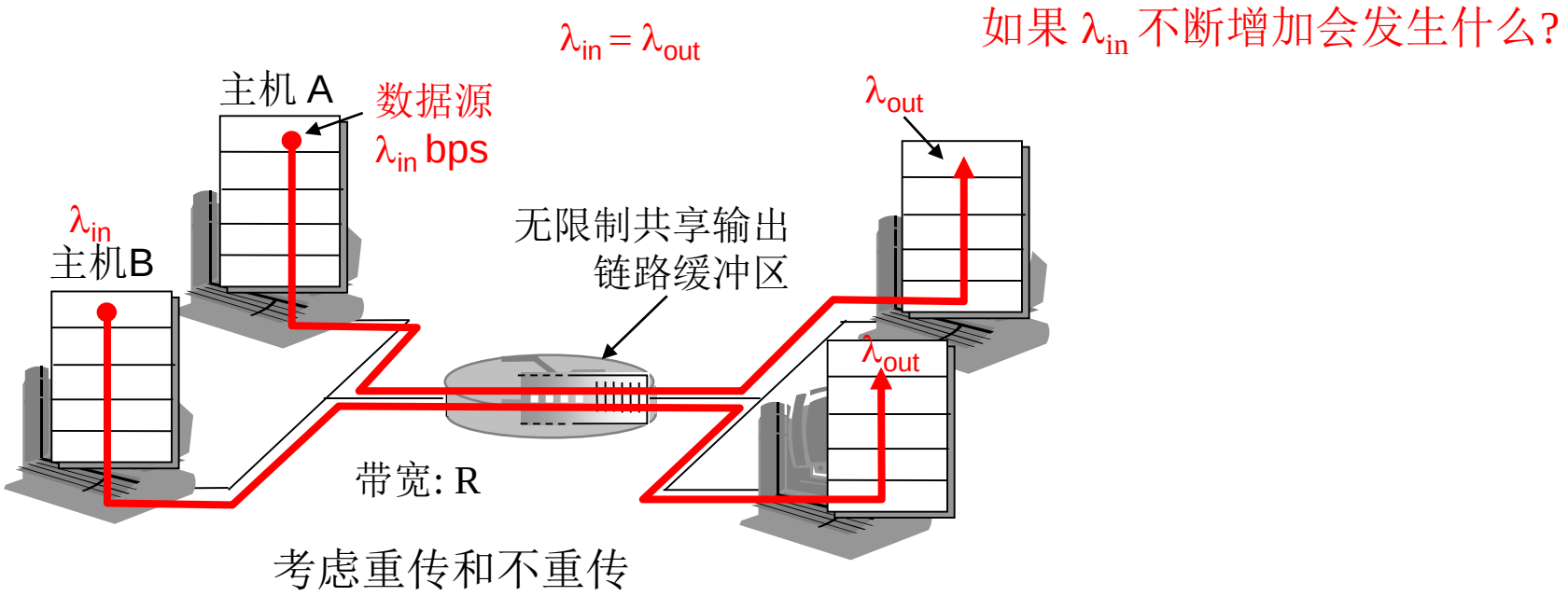
不同操作系统初始值可以不同

在12次重传后TCP协议发送rst数据段并关闭连接。

拥塞的表现

拥塞(a top-10 problem!): 多对一传

- 简单来说: "太多的主机发送太快太多的数据给网络处理"。 这不同于流控制! 控制流量
- 表现:
  - 丢包 (路由器上缓冲区溢出)
  - 长延迟 (在路由器缓冲区中排队)



拥塞控制方法

拥塞控制(Congestion Control)的两大类方法:

端到端的拥塞控制:

- 没有来自网络的明确的反馈
- 终端系统通过丢包和延迟推导的拥塞
- TCP协议的方法

网络辅助的拥塞控制:

- 路由器反馈给终端系统
  - 用一个比特指出拥塞发生(SNA, DECbit, TCP/IP ECN, ATM)
  - 向发送方给一个明确的发送速率。

TCP拥塞控制

- 超时或收到3个重复ack就认为丢包了，看作拥塞发生了。
- TCP协议通过减少发送速率来控制拥塞。发送速率与发送窗口大小有关：

  发送速率 rate = SWS(Sending Window Size)/RTT

- 引入拥塞窗口变量CongWin来限制SWS。

  SWS = Min(CongWin, AdvWin)

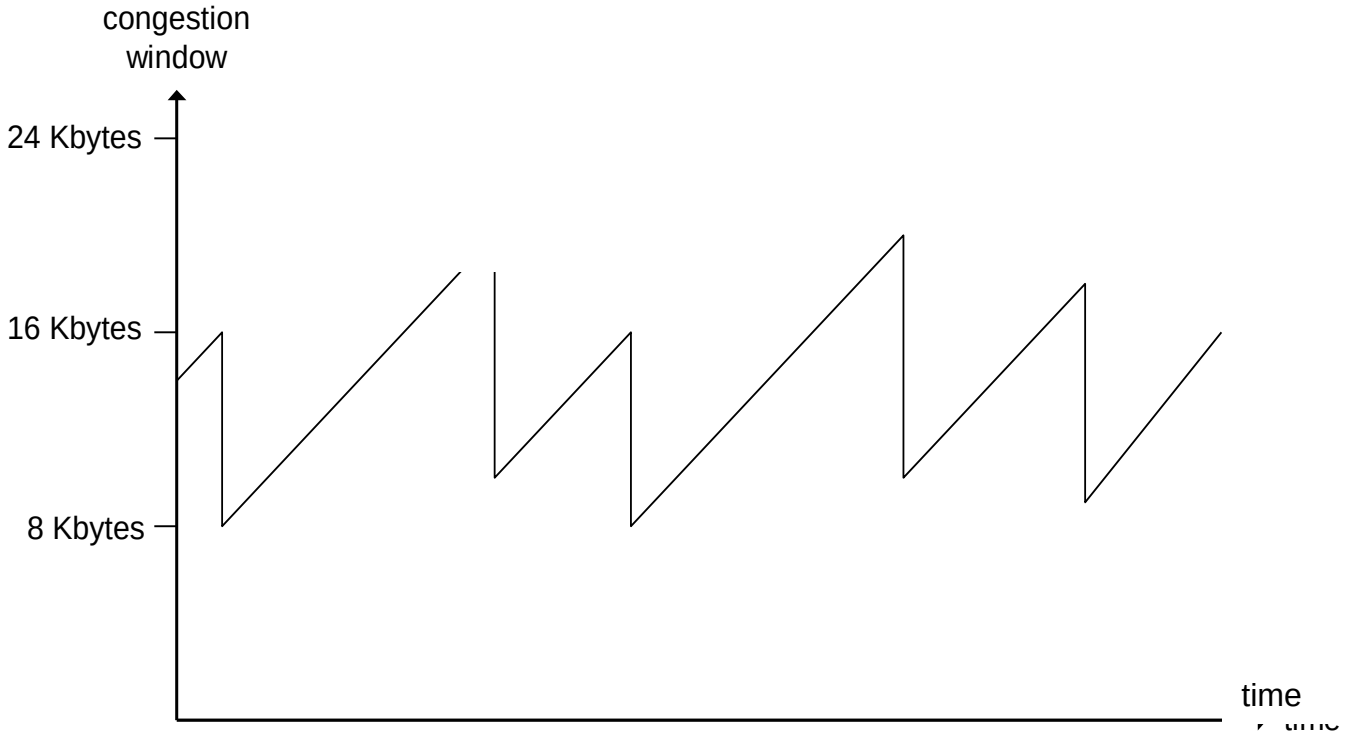
- TCP协议改变CongWin的三种机制：
  - 加性增乘性减(AIMD)
  - 慢启动(slow start)
  - 在超时时间之后的保守方法

CongWin-发送窗口的流量控制
AdvWin-接收窗口要求的流量控制

TCP拥塞控制:加性增乘性减

- *加性增(additive increase):* 每个RTT**CongWin**增加1MSS，直到丢包
- *乘性减(multiplicative decrease):* 在丢包后**CongWin减半**

1st mechanism



用锯齿方式寻找带宽

TCP拥塞控制: 慢启动(slow start)

(1) 初始时，CongWin设为1 MSS, 阈值(threshold)设为65535, 发送一个数据段。

(2) 在当前窗口所有数据段的确认都收到之后, CongWin加倍。
实际上，每收到一个确认，CongWin增加一个MSS。把它称为慢启动(slow start)是因为这个方法比立即采用通知窗口更慢。

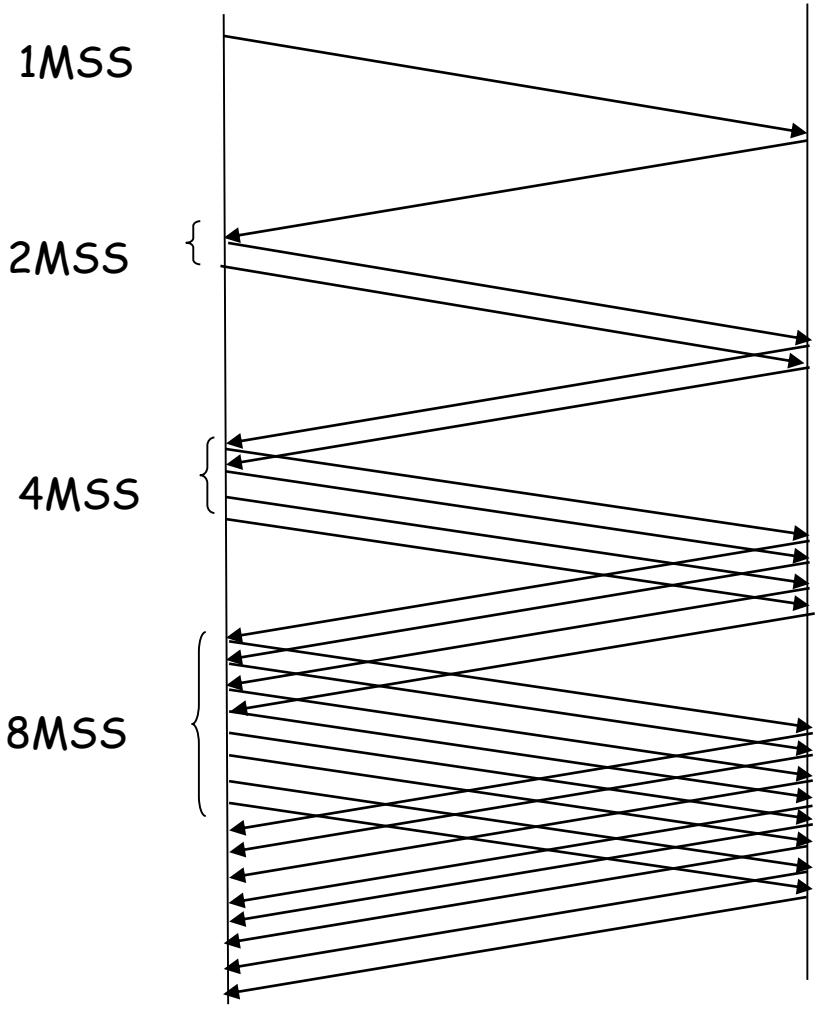
(3) 当拥塞发生时，把当前CongWin (或SWS)的一般保存为阈值(threshold)，然后CongWin又从 1MSS开始慢启动。

(4) 当 CongWin增长到等于或大于阈值 (threshold)时，在当前窗口所有数据段的确认都收到之后, CongWin增加一个MSS。 (Congestion Avoidance)
实际上， 每收到一个ACK，CongWin增加SegSize*SegSize/CongWin 。如果发生拥塞，转 (3)。

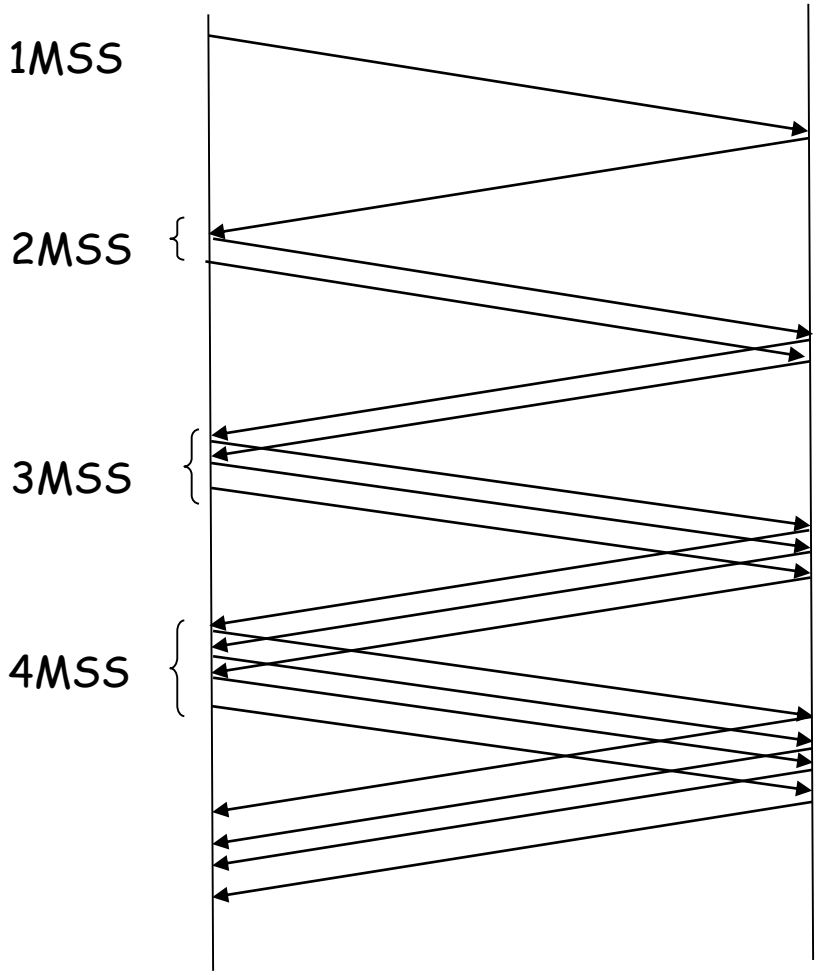
(5) 用系统参数TCP_MaxWin(一般为65535)限制 CongWin的大小。

* SegSize 为被确认的数据段的大小

* 假设算法开始时通知窗口大小AdvWin=65535

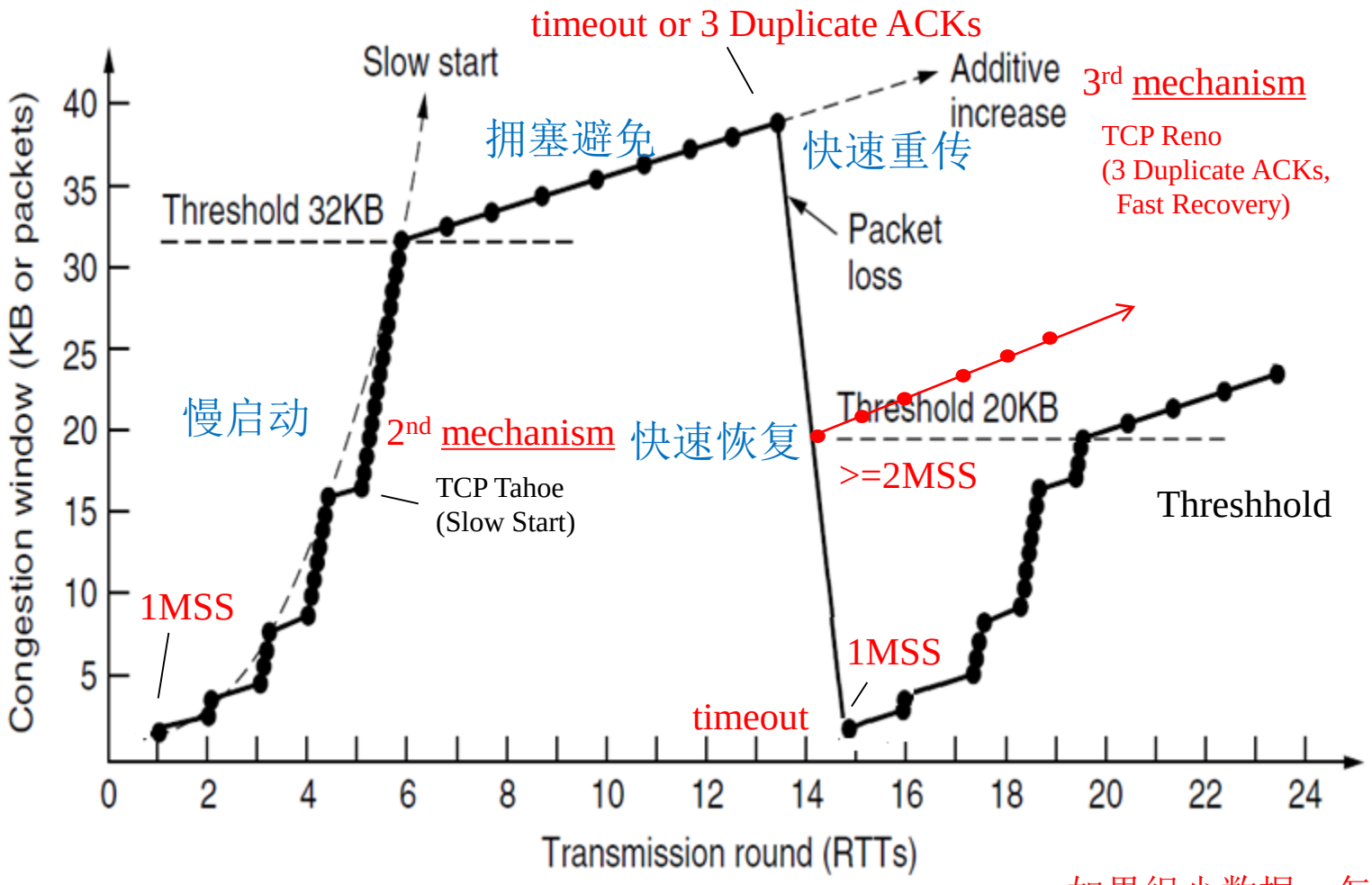


multiplicative(exponential) increase



additive(linear) increase

TCP拥塞控制:
慢启动和快速恢复(Fast Recovery)



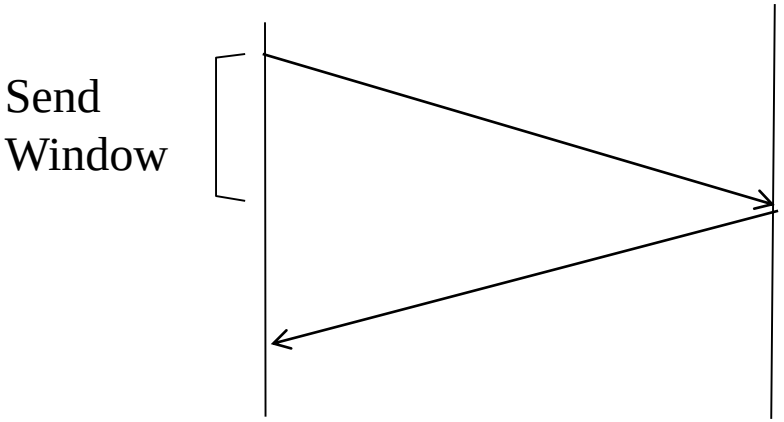
如果很少数据，怎么办？

Slow start followed by additive increase in TCP Tahoe

问题1：长肥管道

如果一个连接被看成是两个端点之间的管道，它的容量（可以容纳的未确认比特数）是多少呢？如果容量很大(long fat pipe), 会出现什么情况？

capacity (bits) = bandwidth (bits/sec) x round-trip-time (sec)



（1）bandwidth= 1Gbps　RTT=16ms，capacity(bytes)= ?

2,000,000 ＞ 通知窗口最大值(65535)　　如何保持管道满载?

（2）bandwidth= 100Gbps RTT =800ms (delayed), capacity(bytes) = ?

10G ＞序号的最大个数(4G)　　如何处理序号回绕?

Window Scaling

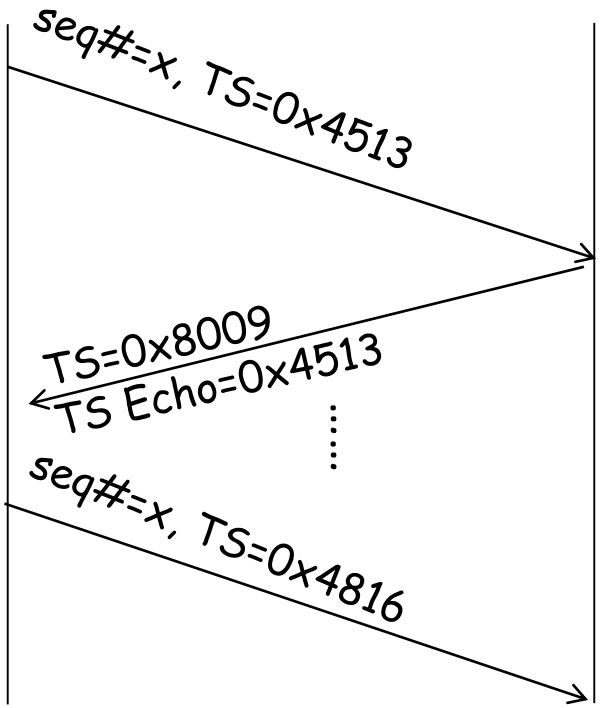
RFC 1323

SYN数据段的选项用于设置WinScale(取值0~14, 默认值为 0)。

Window Size = AdvertisedWindow * $2^{WinScale}$

序号回绕

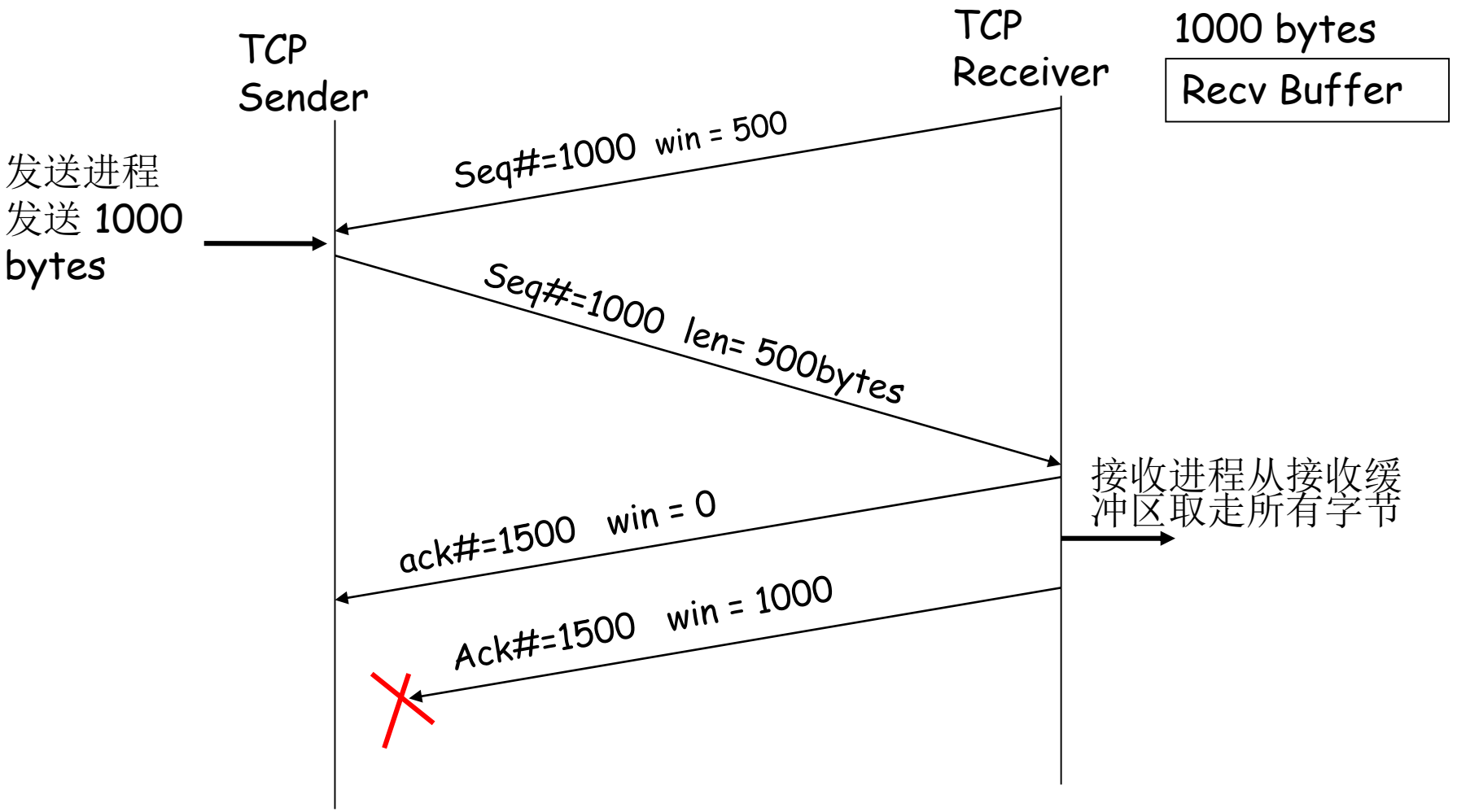
使用一般数据段的选项 "timestamp"(TS)。只用于区分回绕的序号是不同的，不用于确定先后次序。TS也用来测量RTT。



清空管道

当丢包发生时，由于SWS的限制，管道将会被清空。解决方法：快速重传、快速恢复。

问题2：死锁现象

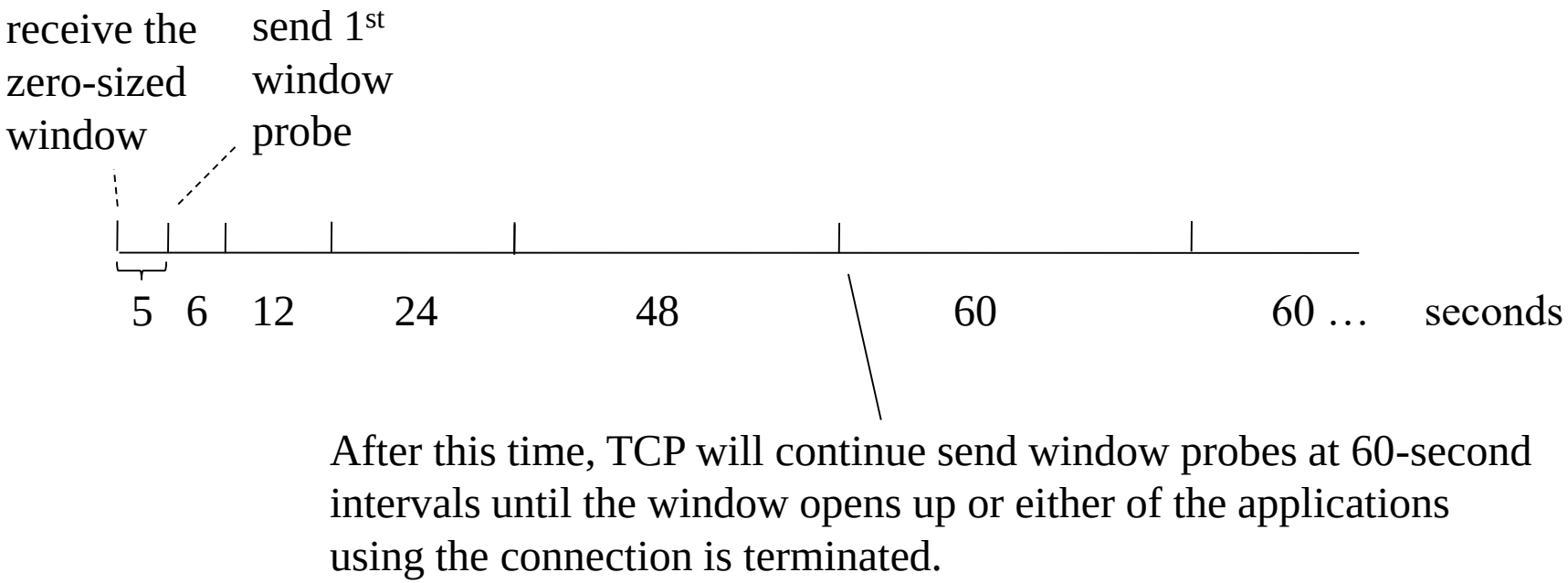


当窗口为0时，如果取空接收缓冲区之后发送的确认丢失，就会发生死锁。

解决方法是什么？

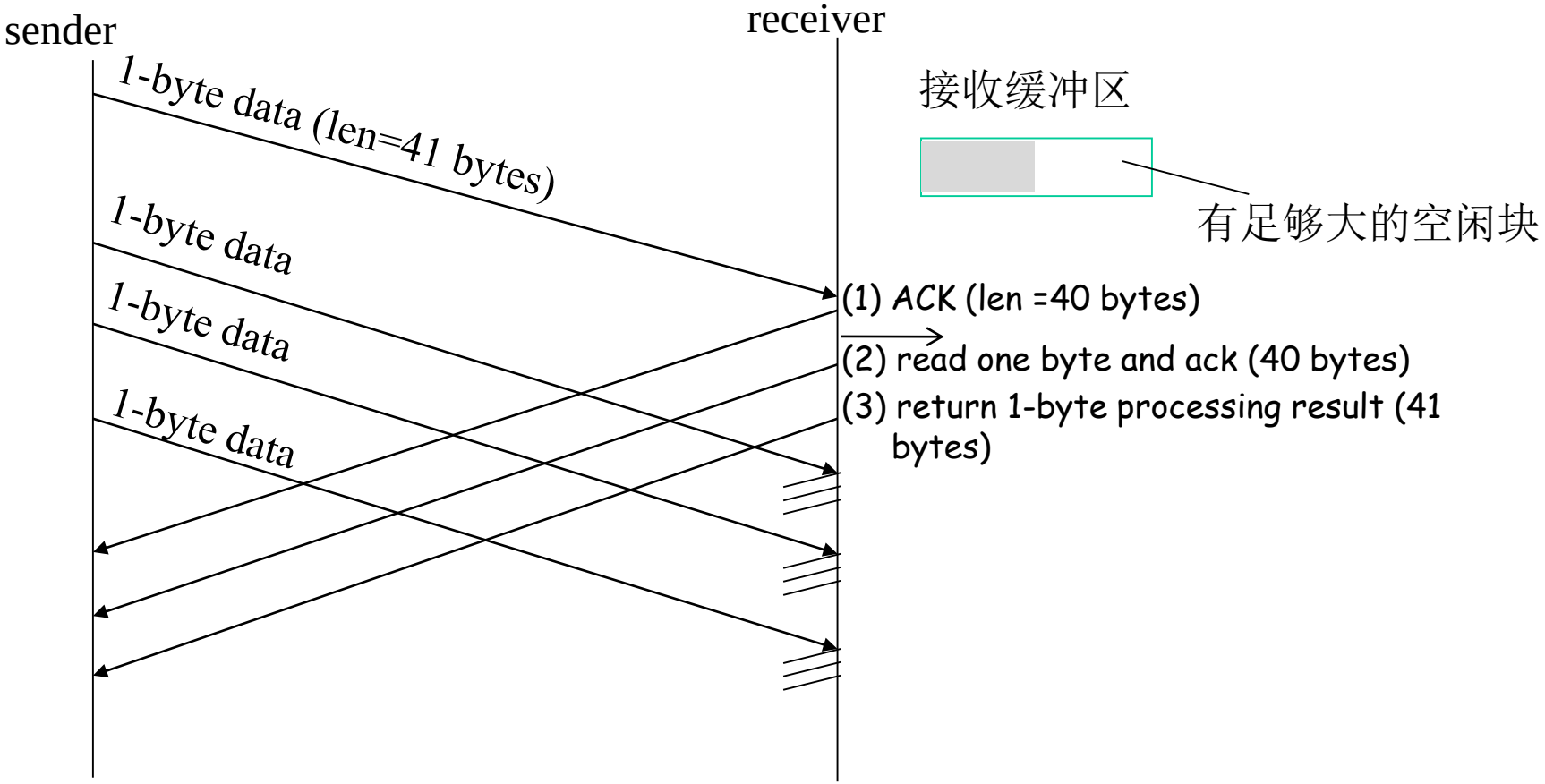
在发送窗口为0之后, 发送方如果有数据要发送，则启动坚持定时器(**Persist Timer**)，定期从要发送的数据中取一个字节发送出去(Window Probe)，直到收到不为0的通知窗口为止。

- Unix



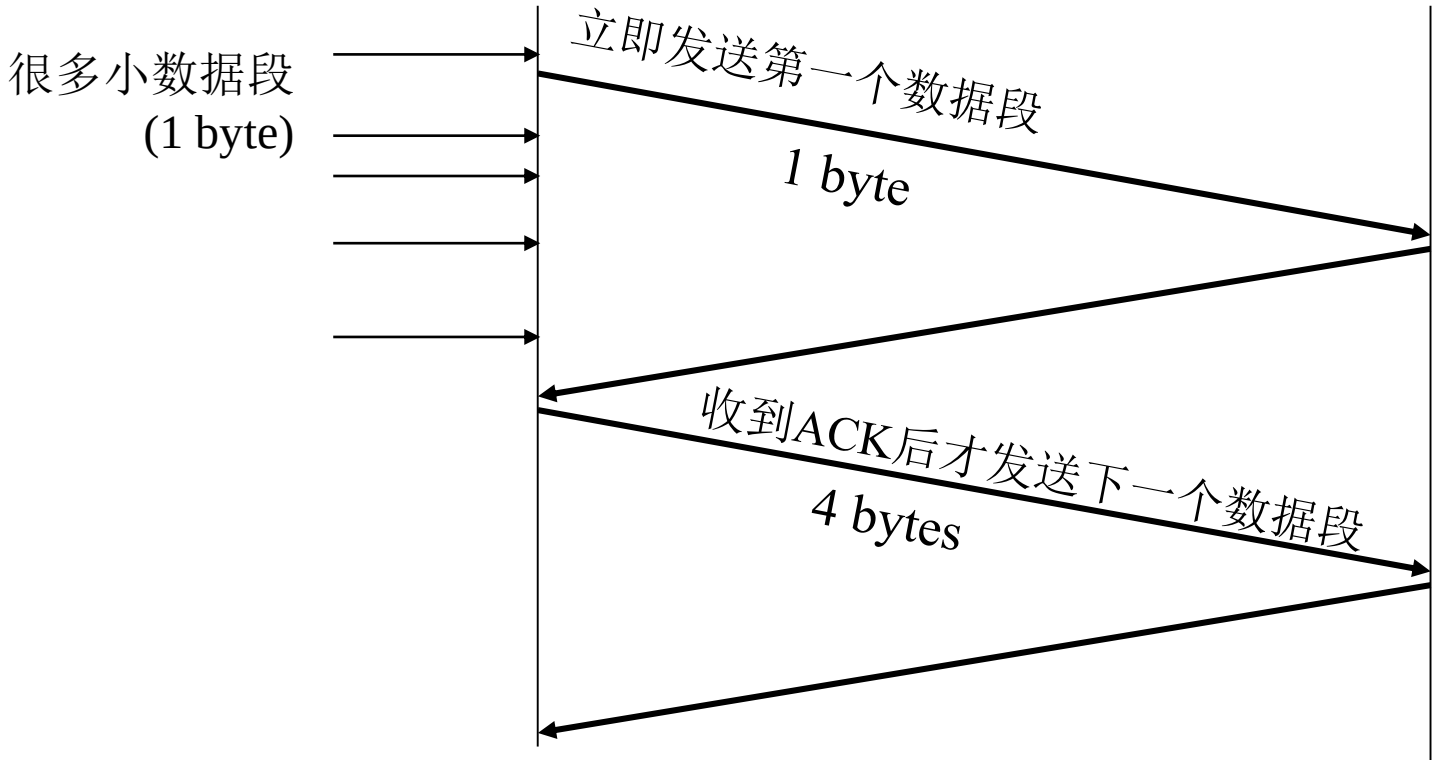
问题3：傻瓜窗口症候（Silly Window Syndrome）

情形1: 发送进程有很多小批量数据要发送, 例如，用telnet作为远程终端。



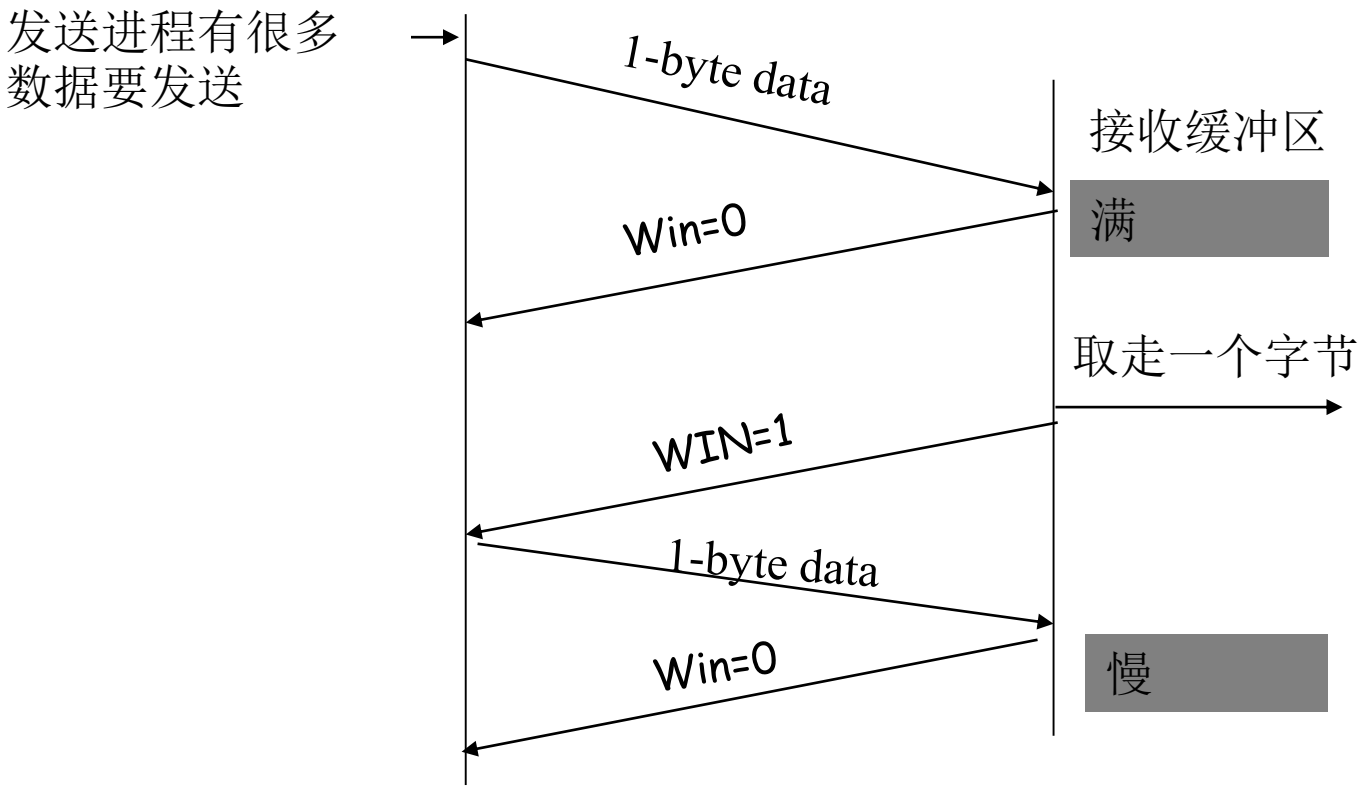
解决方法：收集数据？多少？如何保证及时性?

- Nagle算法---- 启发式算法



(1) 立即发送一个数据段，即使发送缓冲区只有一个字节。
(2) 只有收到上一个数据段的确认或者发送缓冲区中数据超过MSS，才可以发送下一个数据段。
(3) 对于即时性要求高的地方，如，Window方式的鼠标操作，要关闭Nagle算法。

情形2: 接收进程频繁取走小批量数据。



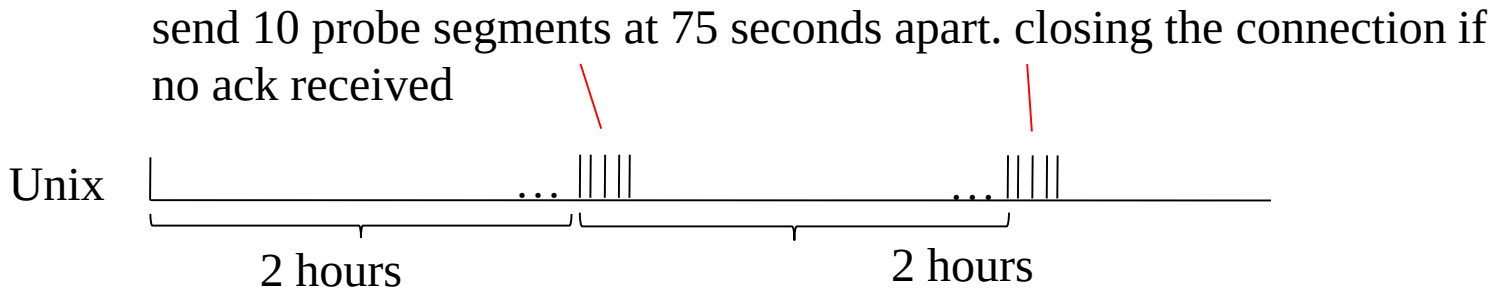
解决方法？当接收缓冲区空闲块 足够大时发送确认。

- Clark算法

当接收缓冲区的空闲块大小变得很小时，要等到空闲块大小为 接收缓冲区 大小 的一半或达到 MSS时才发送确认。

TCP定时器

- 每个连接只针对第一个未确认数据段启动**重传定时**器(retransmission timer)。所有数据段都已确认，则关闭它。超时重传或发送窗口移动时要重启该定时器。
- **持续定时**器(persist timer)用于保持窗口大小信息流动即使连接的另一端关闭了接收窗口。
- **保活定时器**(keepalive timer)在长时间没有交换数据段之后，用于检测连接的另一端是否出了问题。

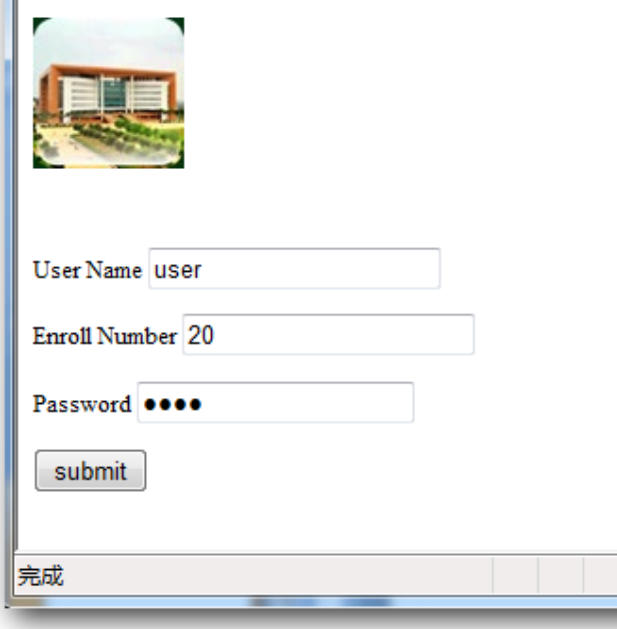


- 处于TIME_WAIT 状态的连接一方需要等待2MSL秒，时间到才能关闭连接。

TCP数据报

```
⊟ Ethernet II
  ⊞ Destination: D-Link_c6:70:7f (00:26:5a:c6:70:7f)
  ⊞ Source: 84:8f:69:a8:07:78 (84:8f:69:a8:07:78)
    Type: IP (0x0800)
⊟ Internet Protocol, Src: 192.168.1.197 (192.168.1.197), Dst: 61.191.56.157 (61.191.56.157)
    Version: 4
    Header length: 20 bytes
  ⊞ Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
    Total Length: 52
    Identification: 0x23fb (9211)
  ⊞ Flags: 0x02 (Don't Fragment)
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
  ⊞ Header checksum: 0x0000 [incorrect, should be 0xddff]
    Source: 192.168.1.197 (192.168.1.197)
    Destination: 61.191.56.157 (61.191.56.157)
⊟ Transmission Control Protocol, Src Port: pacmand (1307), Dst Port: http (80), Seq: 0, Len: 0
    Source port: pacmand (1307)
    Destination port: http (80)
    [Stream index: 7]
    Sequence number: 0    (relative sequence number)
    Header length: 32 bytes
  ⊞ Flags: 0x02 (SYN)
    Window size: 8192
  ⊟ Checksum: 0x38f0 [validation disabled]
      [Good Checksum: False]
      [Bad Checksum: False]
  ⊟ Options: (12 bytes)
      Maximum segment size: 1464 bytes
      NOP
      Window scale: 2 (multiply by 4)
      NOP
      NOP
      SACK permitted
```

TCP Example -- HTTP



Source	Destination	Protocol	Info
192.168.1.198	61.191.56.157	TCP	27672 > http [SYN] Seq=0 Win=8192 Len=0 MSS=1460
61.191.56.157	192.168.1.198	TCP	http > 27672 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=
192.168.1.198	61.191.56.157	TCP	27672 > http [ACK] Seq=1 Ack=1 Win=17520 Len=0
192.168.1.198	61.191.56.157	HTTP	GET /ex/r1.htm?x=1&y=2 HTTP/1.1
61.191.56.157	192.168.1.198	HTTP	HTTP/1.1 200 OK (text/html)
192.168.1.198	61.191.56.157	HTTP	GET /ex/images/campus-east.jpg HTTP/1.1
61.191.56.157	192.168.1.198	TCP	[TCP segment of a reassembled PDU]
61.191.56.157	192.168.1.198	TCP	[TCP segment of a reassembled PDU]
192.168.1.198	61.191.56.157	TCP	27672 > http [ACK] Seq=1211 Ack=3180 Win=17520 Ler
61.191.56.157	192.168.1.198	TCP	[TCP segment of a reassembled PDU]
61.191.56.157	192.168.1.198	TCP	[TCP segment of a reassembled PDU]
192.168.1.198	61.191.56.157	TCP	27672 > http [ACK] Seq=1211 Ack=5228 Win=17520 Ler
61.191.56.157	192.168.1.198	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
192.168.1.198	61.191.56.157	TCP	27672 > http [ACK] Seq=1211 Ack=5559 Win=17189 Ler
192.168.1.198	61.191.56.157	HTTP	POST /test.aspx?x=2&y=8 HTTP/1.1 (application/x-w
61.191.56.157	192.168.1.198	HTTP	HTTP/1.1 200 OK (text/html)
192.168.1.198	61.191.56.157	TCP	27672 > http [ACK] Seq=2084 Ack=6376 Win=16372 Ler
192.168.1.198	61.191.56.157	TCP	27672 > http [RST, ACK] Seq=2084 Ack=6376 Win=0 Le

Establish Connection

Get r1.htm

Get campus-east.jpg

Post test.aspx

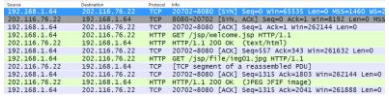
Terminate Connection

* Real SEQ = 0 + ISN

202.116.76.22-Server 192.168.1.64-Client

	Source	Destination	Protocol	Info
1	192.168.1.64	202.116.76.22	TCP	24892→8080 [SYN] Seq=0 Win=65535 Len=0
2	202.116.76.22	192.168.1.64	TCP	8080→24892 [SYN, ACK] Seq=0 Ack=1 Win=81
3	192.168.1.64	202.116.76.22	TCP	24892→8080 [ACK] Seq=1 Ack=1 Win=262144
4	192.168.1.64	202.116.76.22	HTTP	GET /jsp/welcome.jsp HTTP/1.1 （A）
5	202.116.76.22	192.168.1.64	HTTP	HTTP/1.1 200 OK (text/html) （B）
6	192.168.1.64	202.116.76.22	TCP	24892→8080 [ACK] Seq=602 Ack=264 Win=261
7	192.168.1.64	202.116.76.22	HTTP	GET /jsp/file/img01.jpg HTTP/1.1
8	202.116.76.22	192.168.1.64	TCP	[TCP segment of a reassembled PDU]
9	202.116.76.22	192.168.1.64	HTTP	HTTP/1.1 200 OK (JPEG JFIF image)
10	192.168.1.64	202.116.76.22	TCP	24892→8080 [ACK] Seq=1360 Ack=1935 Win=2
11	202.116.76.22	192.168.1.64	TCP	8080→24892 [FIN, ACK] Seq=1935 Ack=1360
12	192.168.1.64	202.116.76.22	TCP	24892→8080 [ACK] Seq=1360 Ack=1936 Win=2
13	192.168.1.64	202.116.76.22	TCP	24892→8080 [FIN, ACK] Seq=1360 Ack=1936
14	192.168.1.64	202.116.76.22	TCP	[TCP Retransmission] 24892→8080 [FIN, AC
15	192.168.1.64	202.116.76.22	TCP	[TCP Retransmission] 24892→8080 [FIN, AC
16	192.168.1.64	202.116.76.22	TCP	[TCP Retransmission] 24892→8080 [FIN, AC
17	192.168.1.64	202.116.76.22	TCP	[TCP Retransmission] 24892→8080 [FIN, AC
18	192.168.1.64	202.116.76.22	TCP	[TCP Retransmission] 24892→8080 [FIN, AC
19	192.168.1.64	202.116.76.22	TCP	24892→8080 [RST, ACK] Seq=1361 Ack=1936

服务器在很短的一段时间内没有再收到http请求，则会发送FIN包并关闭连接。



Comparison of UDP and TCP

TCP service:

- *connection-oriented:* setup required between client and server processes
- *reliable transport* between sending and receiving process
- *flow control:* sender won't overwhelm receiver
- *congestion control:* throttle sender when network overloaded
- *does not provide:* timing, minimum bandwidth guarantees

UDP service:

- unreliable data transfer between sending and receiving process
- does not provide: connection setup, reliability, flow control, congestion control, timing, or bandwidth guarantee

Q: why bother? Why is there a UDP?

总结

- 概述
- TCP/UDP报文
- 传输层的多路复用
- 端口号
- UDP协议
- 传输控制协议(TCP)
- TCP报文格式
- 三次握手建立连接
- TCP状态转换图
- TCP滑动窗口
- 快速重传
- 延迟确认
- 选择性确认
- TCP超时计算
- 拥塞控制
- 问题1：长肥管道
- 问题2：死锁现象
- 问题3：傻瓜窗口症候
- TCP定时器