第六章 传输层

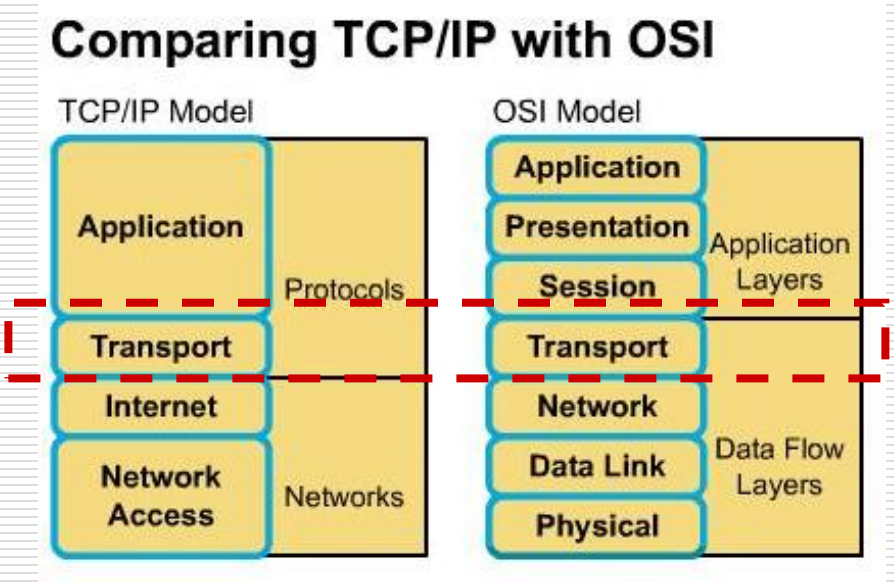
袁华：hyuan@scut.edu.cn

华南理工大学计算机科学与工程学院

广东省计算机网络重点实验室

本章的主要内容

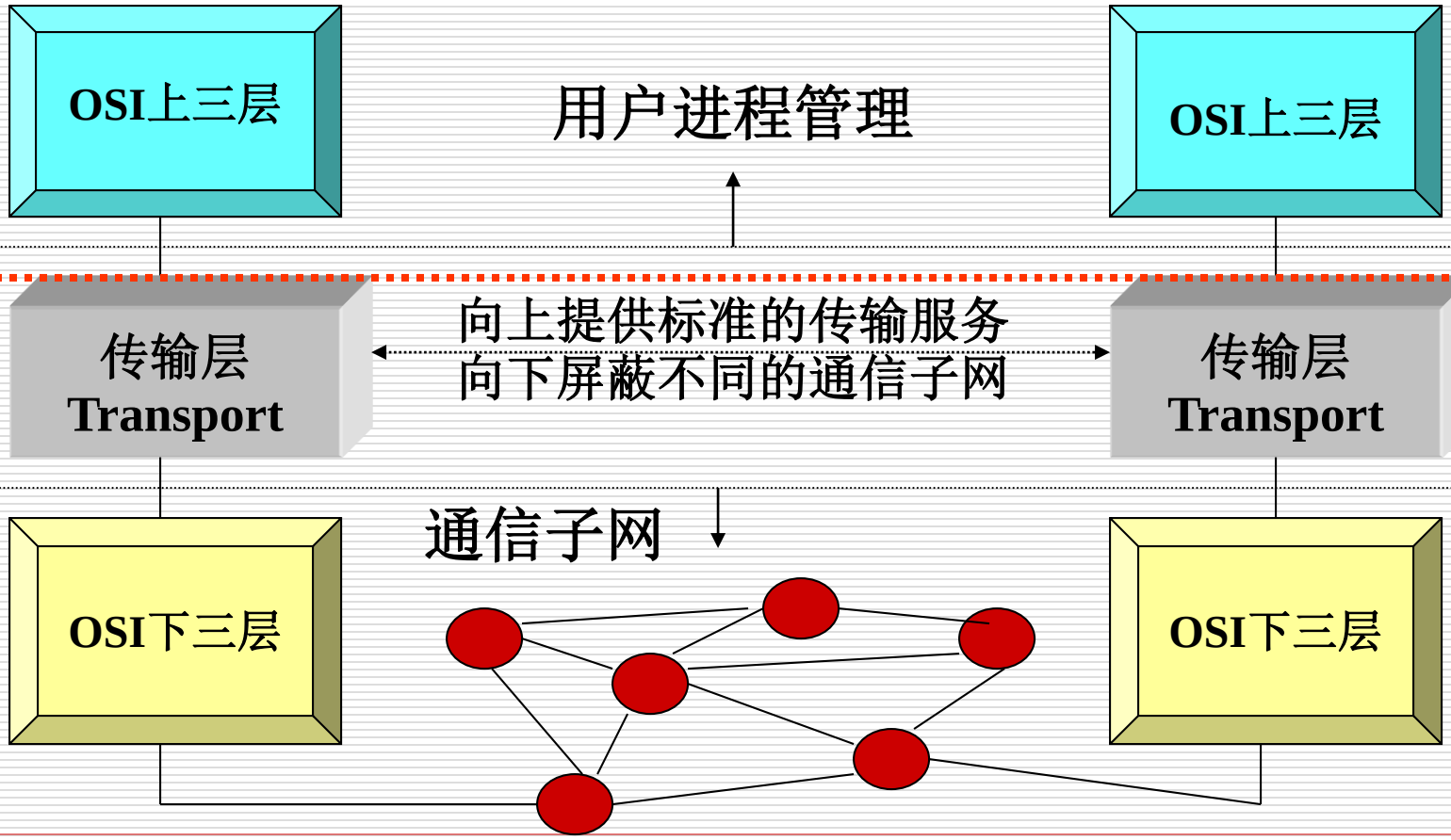
- 传输层概述
- 传输层协议
  - UDP
  - TCP
    - 段格式
    - 连接建立 (三次握手)
    - 连接释放
    - 怎样提供可靠的数据传输?
- Socket编程



传输层概述P382

- 传输层是整个协议栈(TCP/IP)的核心
- 传输层的任务是提供可靠的、高效的数据传输
- 本章讨论传输层的服务、设计、协议和性能等（ services, design, protocols, and performance ）

传输层的地位 P383

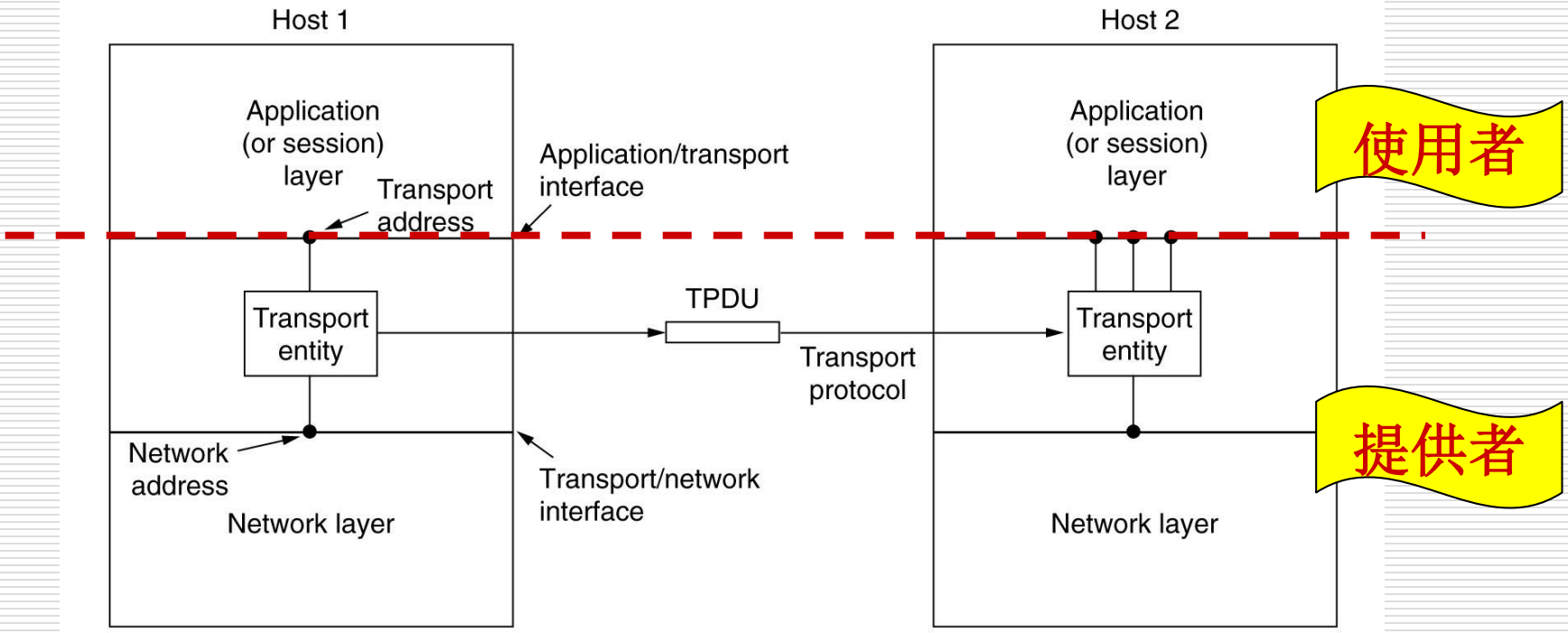


传输服务用户

传输服务提供者

网络层、传输层和应用层P382

- 传输层在应用层和网络层之间提供了无缝接口
- 下四层被看作传输服务提供者，而上三层是传输服务使用者



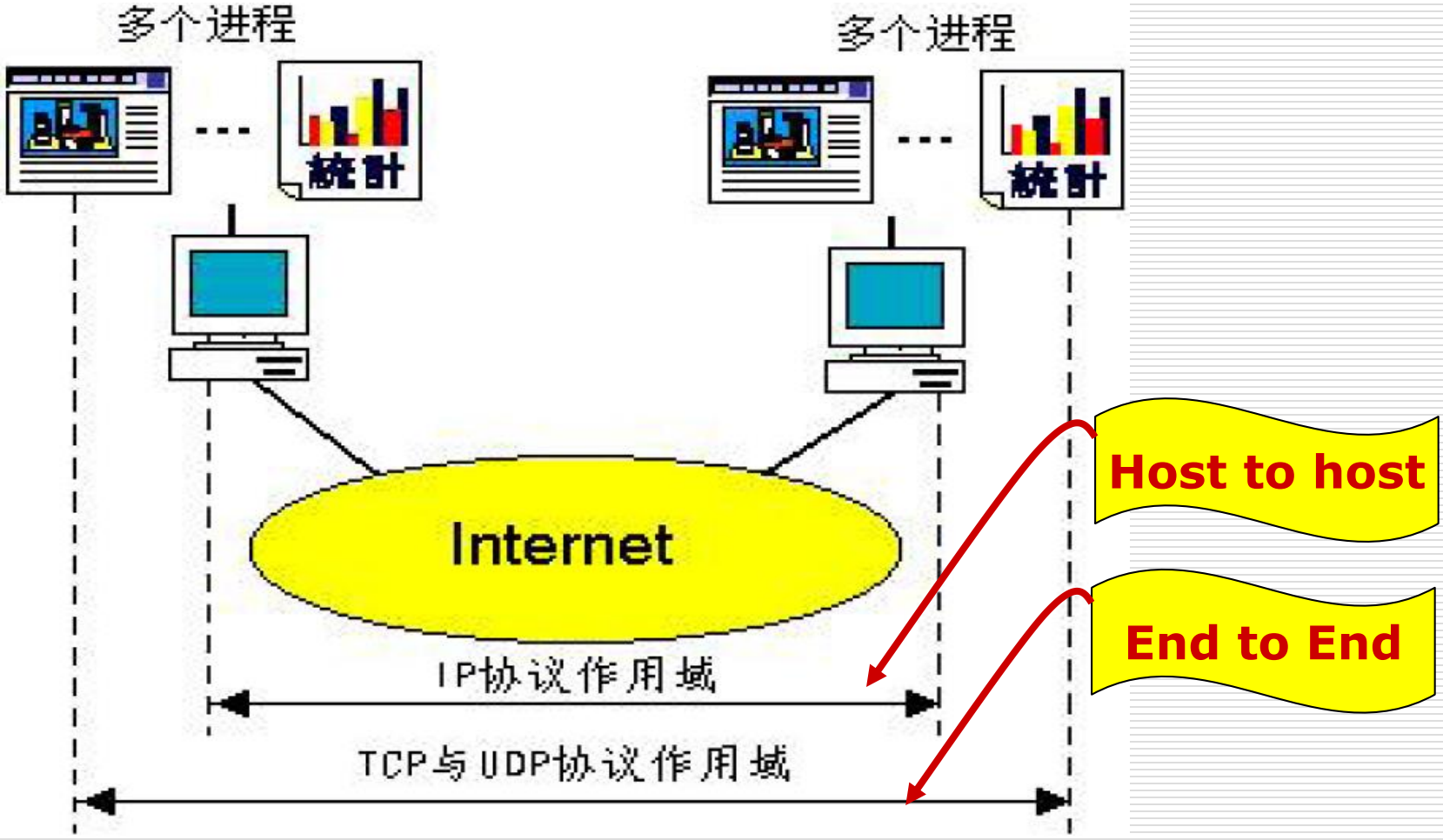
向上层提供服务P383

- 传输层的最终目标是向它的用户（应用层）提供高效、可靠和性价比高的服务
- 完成这项工作的硬件或软件被称为传输实体（transport entity）
- 传输实体可能位于：
  - 操作系统内核
  - 独立的用户进程中
  - 绑定在网络应用中的链接库
  - 网络接口卡
  - 。。。。。。。

传输层服务P383

- 有两种传输层服务：
  - 面向连接的服务
  - 无连接的服务Connectionless
- 这和网络层提供的服务相似，那为什么需要两个独立的不同的层？
  - 网络层运行在由承运商操作的路由器上，因此用户无法真正控制到网络层
  - 把另一层放在网络层之上，可以让用户能够控制到服务质量（**some control**）.
  - 传输层原语独立于网络层原语，而网络层原语会因为网络的不同而不同

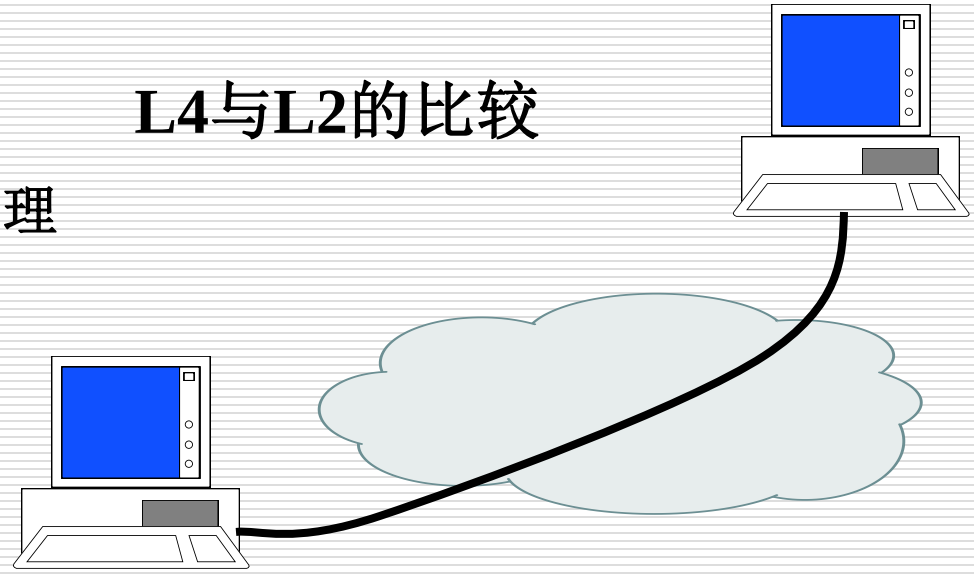
作用范围的比较


多个进程
多个进程
统计
统计
Internet
IP协议作用域
TCP与UDP协议作用域
Host to host
End to End

传输层提供的功能

- 端点标识
- 传输服务
  - 面向连接
    - 端到端的连接管理
      - 建立连接
      - 数据传输
      - 释放连接
    - 流控制
    - 差错控制
  - 无连接

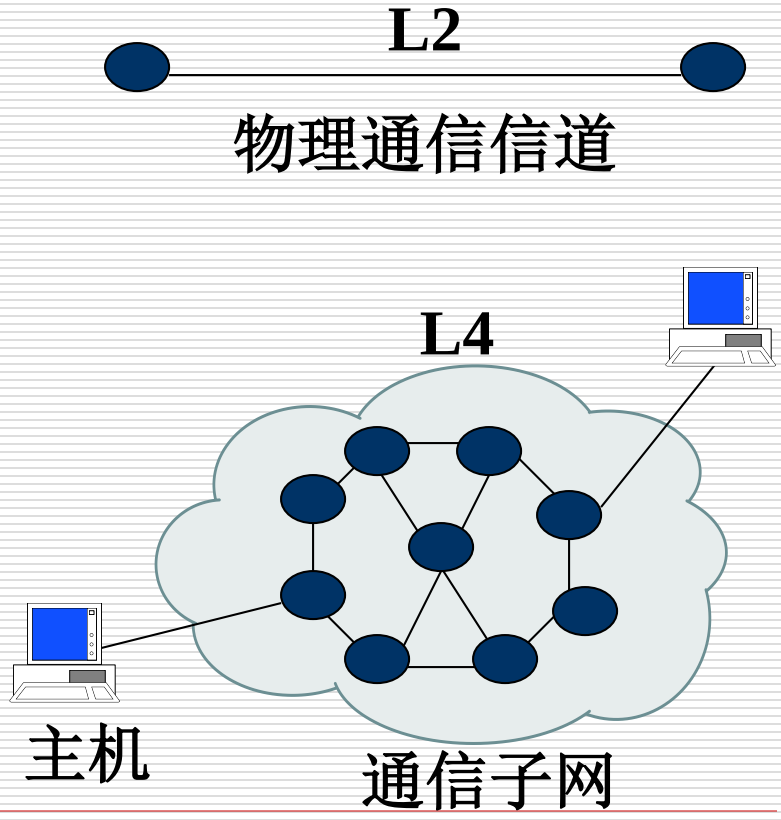
L4与L2的比较



L4与L2的比较

针对面向连接服务：

- 功能相似
  - 流控制
  - 差错控制
  - 数据（报文/帧）排序
- 传输环境不同
  - 通信子网
  - 物理信道



传输服务原语P383

- 传输服务原语让应用程序可以有途径访问到传输服务
- 传输服务和网络服务的两个主要差别是：
  1. 网络服务试图按照实际网络提供的服务来建模（不可靠的）；而面向连接的传输服务是可靠的
  2. 网络服务仅被传输实体所使用；而传输服务直接被应用程序所使用，必须方便易用

简单的传输服务原语 P384

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Actively attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

使用传输服务原语的一个例子（1/3）

- 连接建立：
  - 服务器调用 LISTEN 原语，阻塞该服务器，直到有客户来连接
  - 当一个客户想和服务器通信，它调用 CONNECT 原语，阻塞该客户，且发送一个分组（封装了传输消息）
  - 当分组到达服务器端，传输实体检查看是否服务器阻塞在 LISTEN 调用中，然后，他解除阻塞，并向客户发回一个CONNECT ACCEPT数据段
  - 当这个数据段到达客户端的时候，客户端解除阻塞，连接建立

使用传输服务原语的一个例子（2/3）

□ 数据交换

- 任何一方都可以执行阻塞的 **RECEIVE** 原语，以等待另一方执行**SEND**原语
- 当数据数据段（ **TPDU**）到达时，接收方解除阻塞，并对这个数据段进行处理，发回一个应答
- 只要双方对数据的认识有统一的认识，这种机制可以工作得很好
- 每个发出的分组都要被确认，利用网络层的服务；这些确认、定时器、重传等，由传输实体使用网络层协议来管理，对传输层的用户来说，这些都是不可见的

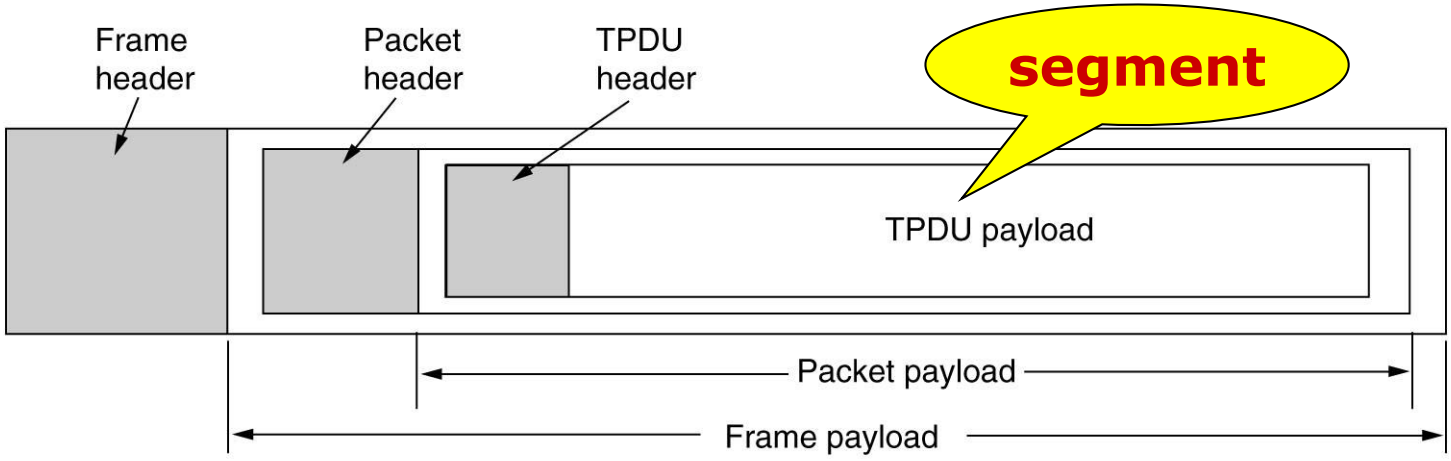
使用传输服务原语的一个例子（3/3）

连接释放

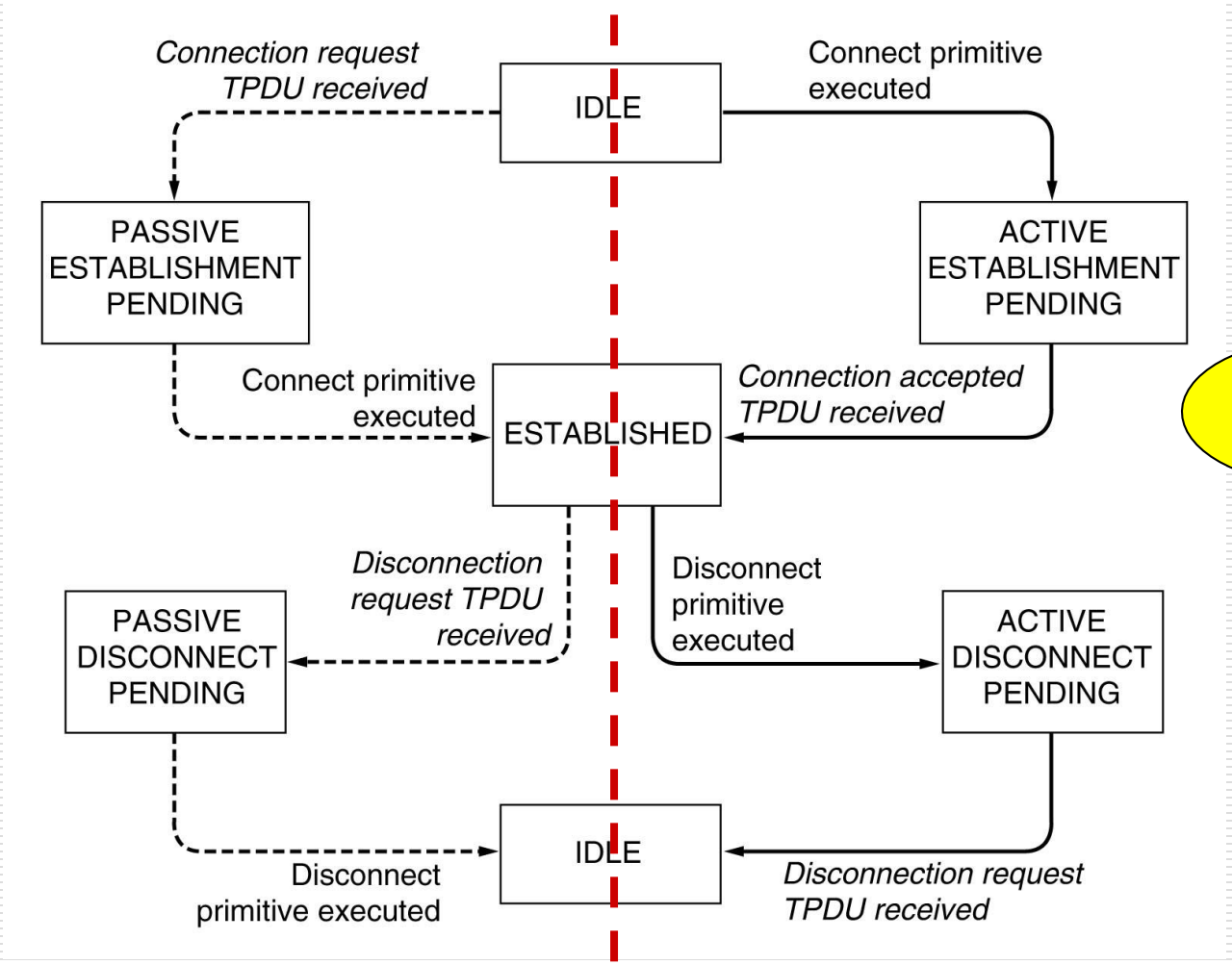
- 非对称的释放： 任何一方都可执行 **DISCONNECT** 原语，将 **DISCONNECT**数据段发给对端的传输实体，数据段到达另一端，连接被释放
- 对称的释放： 当一方执行 **DISCONNECT**原语的时候，这意味着它没有更多的数据要发了，但是仍然希望接收数据，只有当双方都执行 **DISCONNECT** 原语后，一个连接才真正被释放

数据段（TPDU，传输层协议数据单元）

- **TPDU (Transport Protocol Data Unit) 是从传输实体发到对端传输实体的信息P385**
- **TPDUs 被封装在分组（packet）中，由网络层交换**
- **分组被封装在帧（frames）中，由数据链路层交换**



一个简单的连接管理方案P386



服务器

客户端

传输层协议

- **UDP(6.4)**
  - **User datagram protocol**
- **TCP(6.5)**
  - **Transport control protocol**

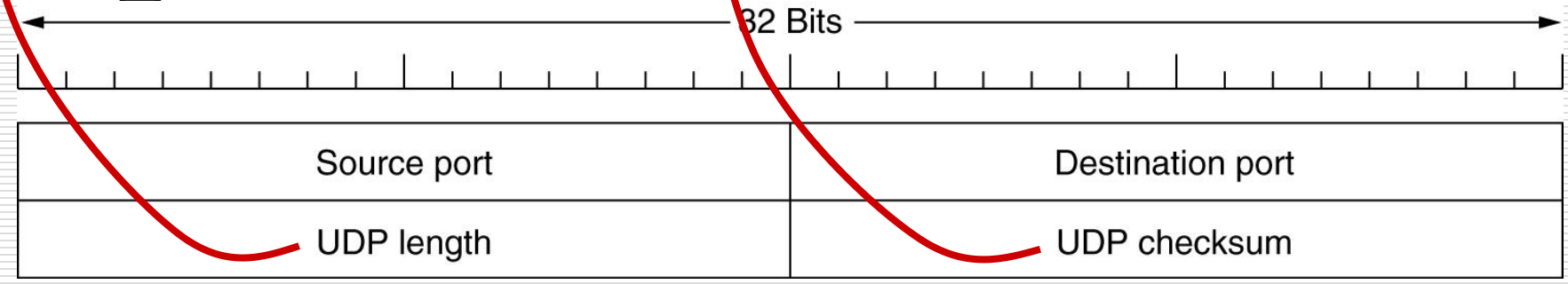
User Datagram Protocol (6.4)P417

- UDP 是一个无连接的（connectionless）的传输层协议
- UDP传输数据段，无须建立连接
- UDP 在 RFC 768中描述
- 很多C/S应用(如： DNS)，都使用UDP发送一个请求， 然后对方应答

为什么需要 UDP?

UDP 数据段头 P418

- UDP 数据段包括8字节（ 8-Byte）的头部和数据两个部分
- 其中的长度域表示的长度包括头部和数据总共的长度
- 校验和（checksum）是可选的，如果不计算校验和，则该域置为 0
- UDP比IP好的地方在于它可以使用源端口和目的端

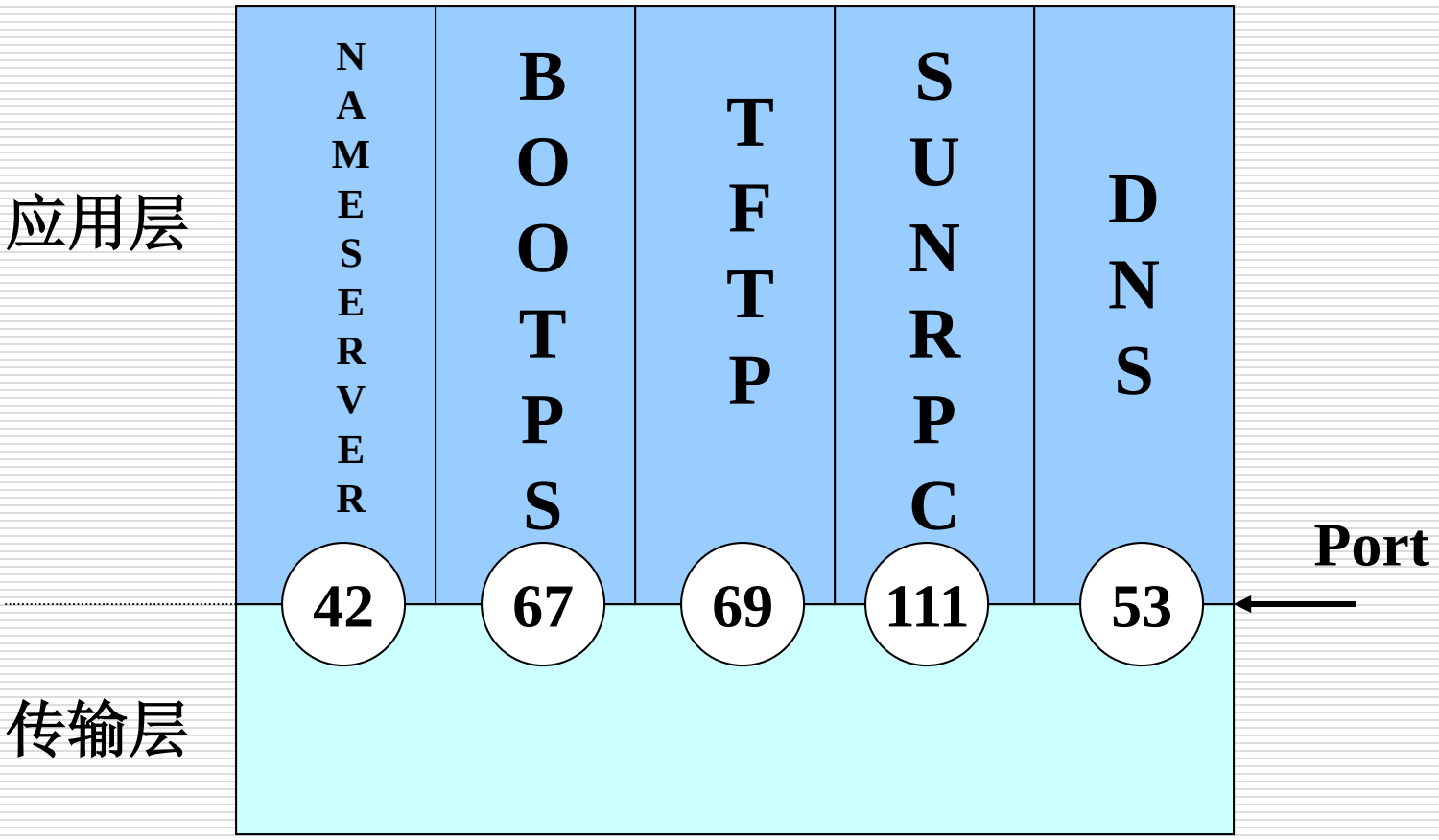


端口（port）定义 P426

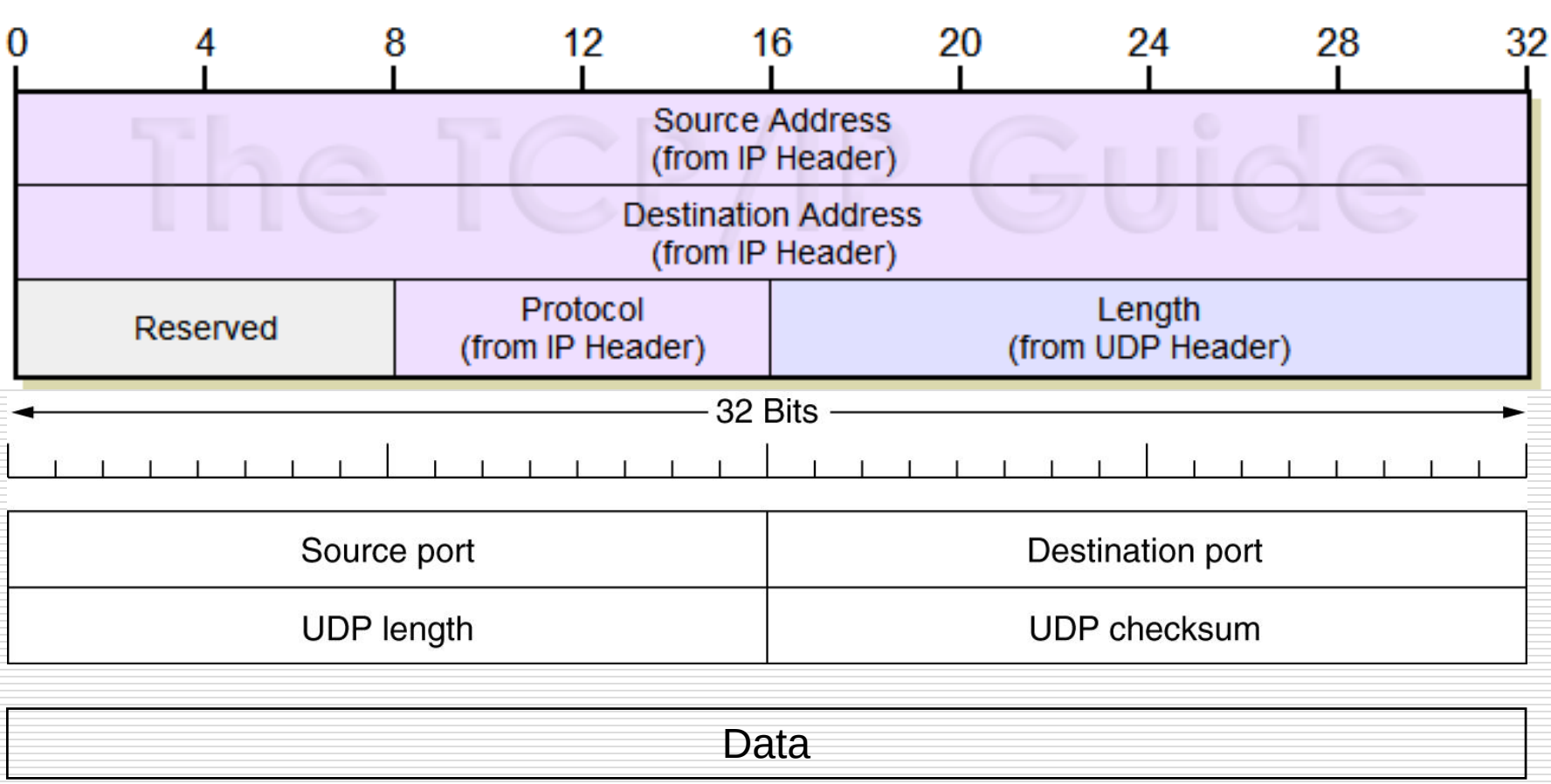
- 16 位，共有 2^{16} 个端口
  - 端口范围：0~65535
    - <1023 : 用于公共应用（保留，全局分配，用于标准服务器），IANA分配；
    - 1024~49151 :用户端口，注册端口；
    - >49152 : 动态端口，私人端口。
- 自由端口(Free port)
  - 本地分配
  - 动态的随机端口

RFC 6335

UDP 保留端口


应用层
NAMESERVER
BOOTPS
TFTP
SUNRPC
DNS
Port
42
67
69
111
53
传输层

UDP 校验和 P418


0
4
8
12
16
20
24
28
32
Source Address
(from IP Header)
Destination Address
(from IP Header)
Reserved
Protocol
(from IP Header)
Length
(from UDP Header)
32 Bits
Source port
Destination port
UDP length
UDP checksum
Data

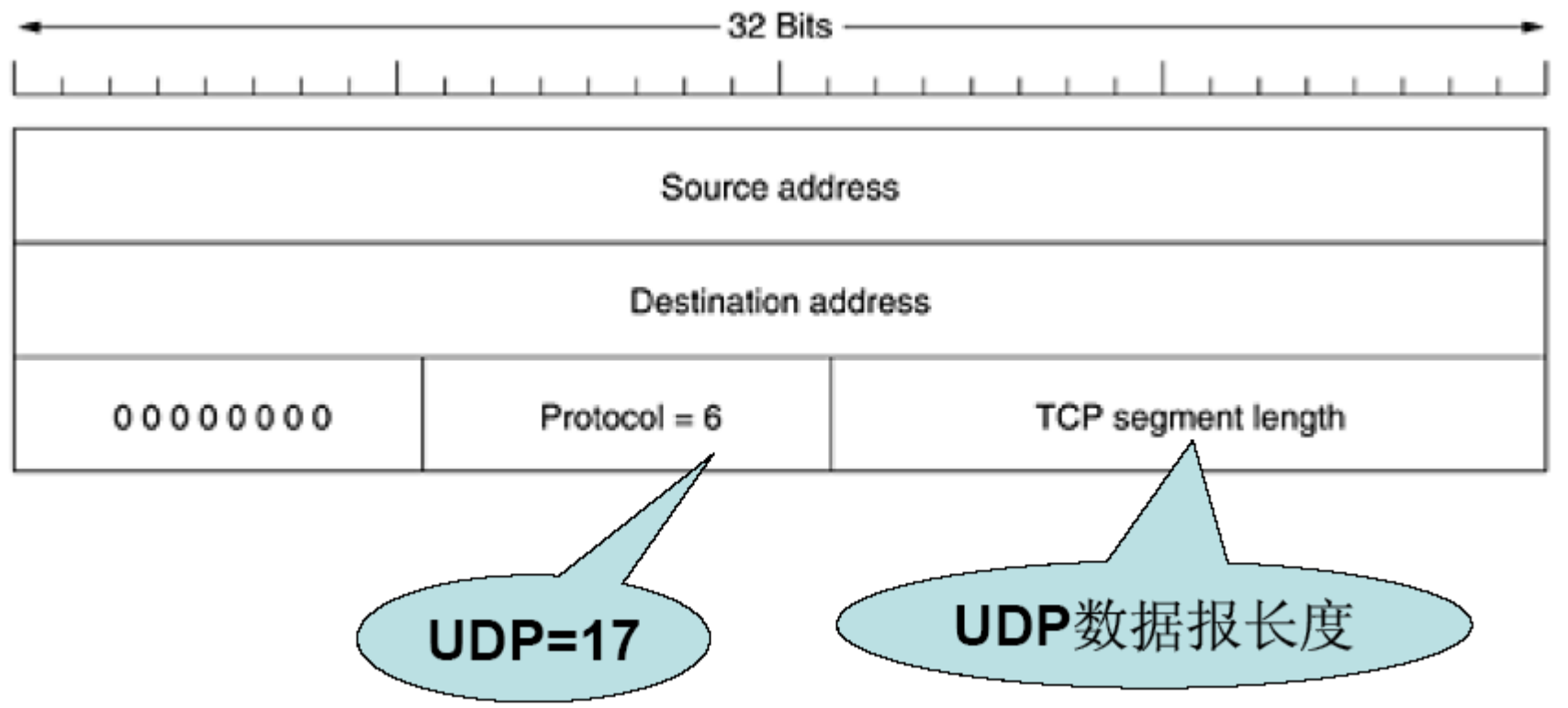
校验和

10011001 00010011	153.19
00001000 01101000	8.104
10101011 00000011	171.3
00001110 00001011	14.11
00000000 00010001	0和17
00000000 00001111	15
00000100 00111111	1087
00000000 00001101	13
00000000 00001111	15
00000000 00000000	0（校验和）
01010100 01000101	数据
01010011 01010100	数据
01001001 01001110	数据
01000111 00000000	数据和填充0

按二进制反码求和：10010110 11101011　　求和得出的结果

将得出的结果求反：01101001 00010100　　校验和

TCP/UDP伪头部(pseudo header)

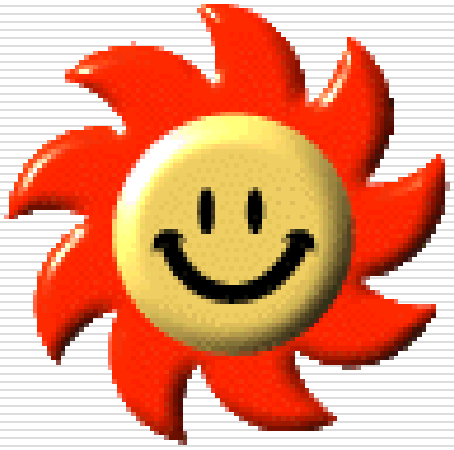


注意

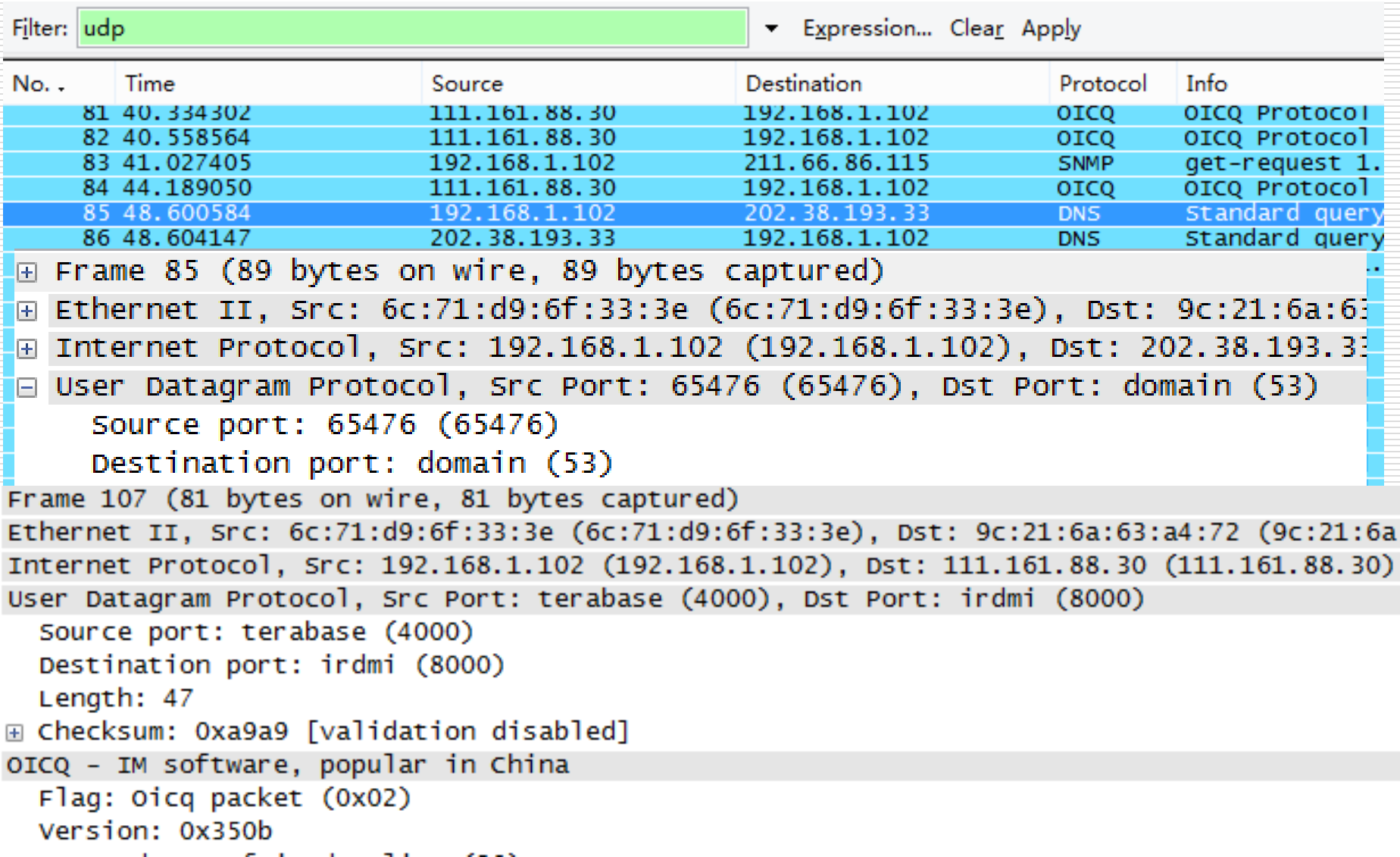
- 如果收方的校验和为全1，传输无错
- 二进制反码求和
  - 从低位到高位逐列计算
  - 0和0相加是0，0和1相加是1，1和1相加是0，但产生进位
  - 最高位相加产生进位，该位为1
- 检错能力较弱，但简单快速
- 使用协议地址，破坏了分层原则

小结：UDP

- 提供端点标识，端到端的数据传输
- 不提供差错检测和可靠传输。但简洁高效



一个UDP数据段的例子



传输控制协议 （6.5 P425）

- TCP (Transmission Control Protocol) 是专门为了在不可靠的互联网络上提供可靠的端到端字节流而设计的
- TCP必须动态地适应不同的拓扑、带宽、延迟、分组大小和其它的参数，并且当有错误的时候，能够足够健壮

传输控制协议 (续)

- 支持TCP的机器都有一个 TCP 实体，或者是用户进程或者是操作系统内核，都可以管理TCP流和跟IP层的接口P426
  - TCP实体接收本地进程的用户数据流，将其分割成不超过64kB的分片（实践中，通常分割成1460字节，以通过以太网传输）
  - 当包含TCP数据段的报文到达某台机器的时候，被提交给传输实体，传输实体将其重构出原始的字节流

发：封装

收：解封装

TCP 服务模型6.5.2 P427

- 要想获得TCP服务，发送方和接收方必须创建一种称为套接字（ **sockets** ）的端点（ **end points**）
- 每个套接字是包含一个IP地址和一个16位的端口（ **port** ）
- 通信进程的全球唯一标识
  - 三元组：协议、本地地址、本地端口号
  - 五元组：协议、本地地址、本地端口号、远端地址、远端端口号

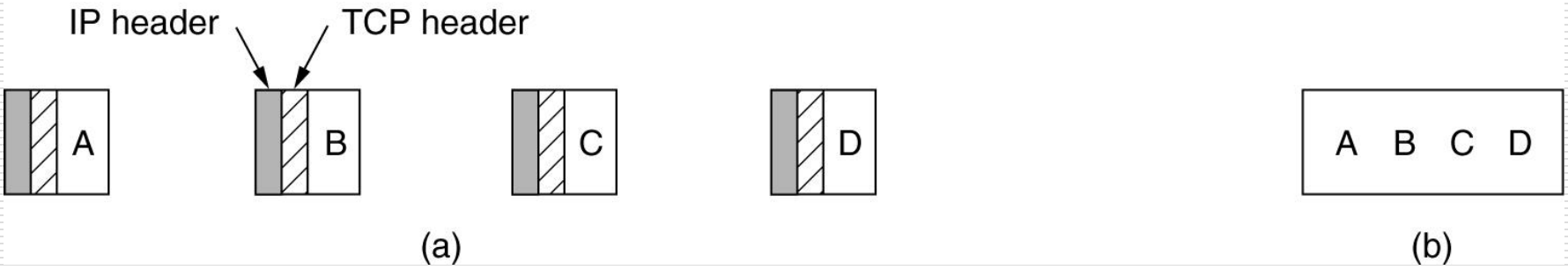
一些已分配的知名端口

- **The list of well-known ports is given at http://www.iana.org .**

Port	Protocol	Use
21	FTP	File transfer
23	Telnet	Remote login
25	SMTP	E-mail
69	TFTP	Trivial file transfer protocol
79	Finger	Lookup information about a user
80	HTTP	World Wide Web
110	POP-3	Remote e-mail access
119	NNTP	USENET news

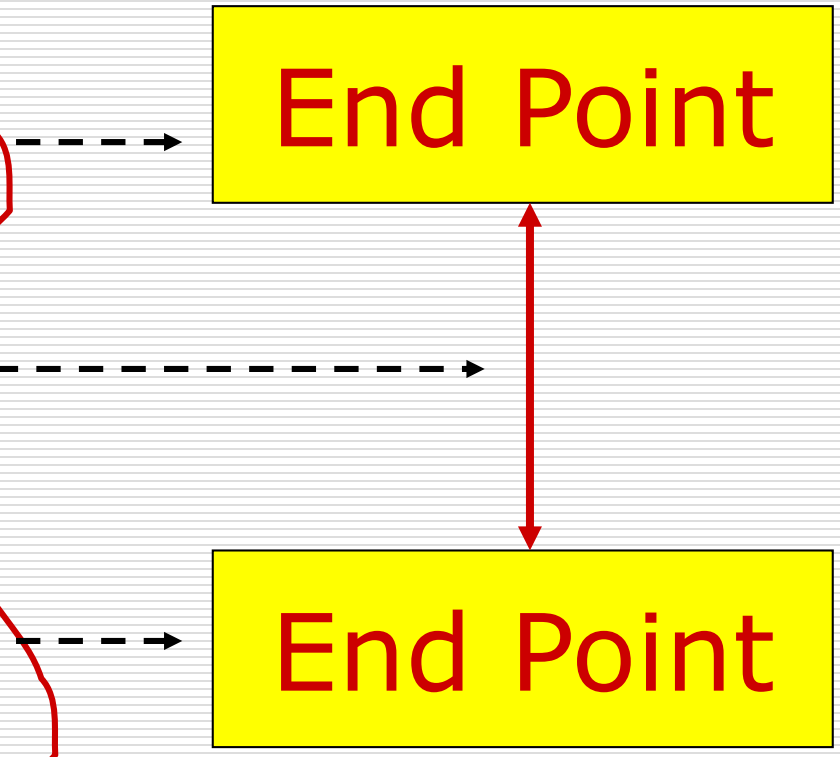
TCP 服务模型 (P427)

- 所有的 TCP 连接是全双工的(同时双向传输)和端到端的(每条连接只有两个端点)
- TCP 不支持组播和广播
- TCP连接是字节流而不是消息流P427



5 元组

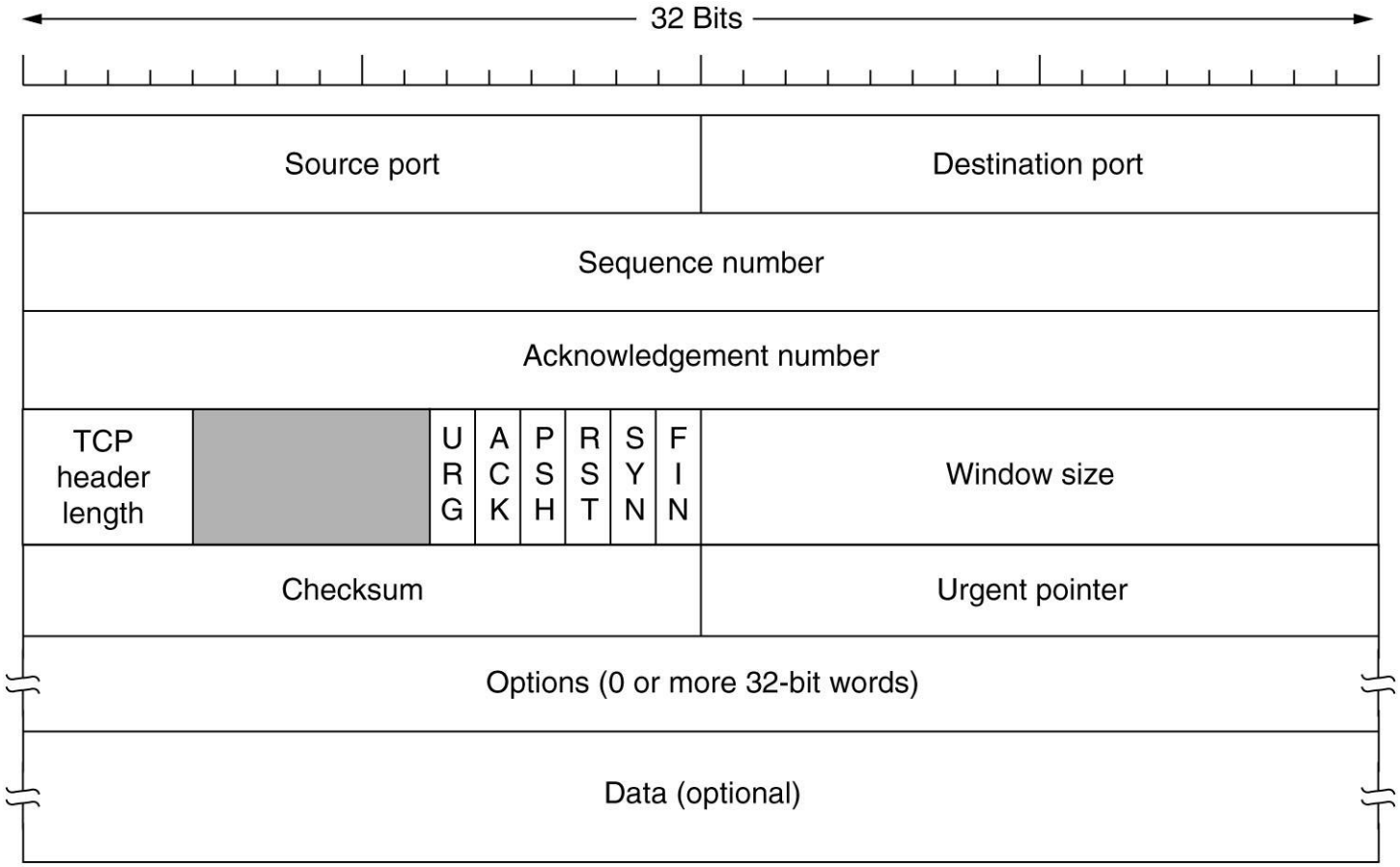
- Source IP
- Source port
- TCP
- Destination IP
- Destination port



TCP 协议 6.5.3 P428

- TCP连接上的每个字节都有它自己独有的32位序列号
- 收发双方的TCP实体以数据段的形式交换数据
- 一个数据段包括20字节的头部（不包括可选项）和数据域（0或更多字节）
- TCP软件决定数据段的大小，有两个因素限制了数据段的长度：
  - TCP数据段必须适合IP的65515（？）的载荷限制
  - 每个TCP数据段必须适合于下层网络的 MTU （如, 1500 字节 – 以太网载荷大小）
- TCP使用的基本协议具有动态窗口大小的滑动窗口协议（sliding window protocol） P429

TCP 数据段（TPDU）格式 P429

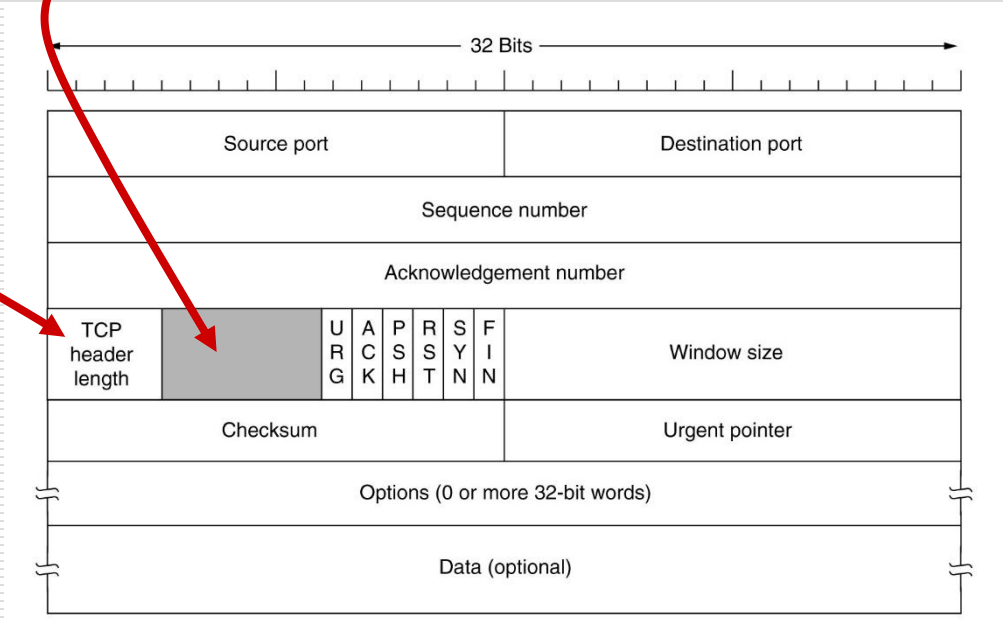


TCP 数据段头 （6.5.4 P429）

- 源端口 和 目的端口 字段标明了一个连接的两个端点
  - 用来跟踪同一时间内通过网络的不同会话。一般每个端口对应一个应用程序
- 序列号 – 字节号 (32 位)
  - 初始序列号ISNs(initial sequence numbers)：随机产生的
  - SYN: 携带了ISNs 和SYN 控制位的数据段
- 确认号 – 期望接收的字节号 (32位)

TCP 数据段头(续)

- **TCP 段头长度 – TCP段头长度，单位32位（4字节）**
- **保留域/字段**



TCP数据段头(续)

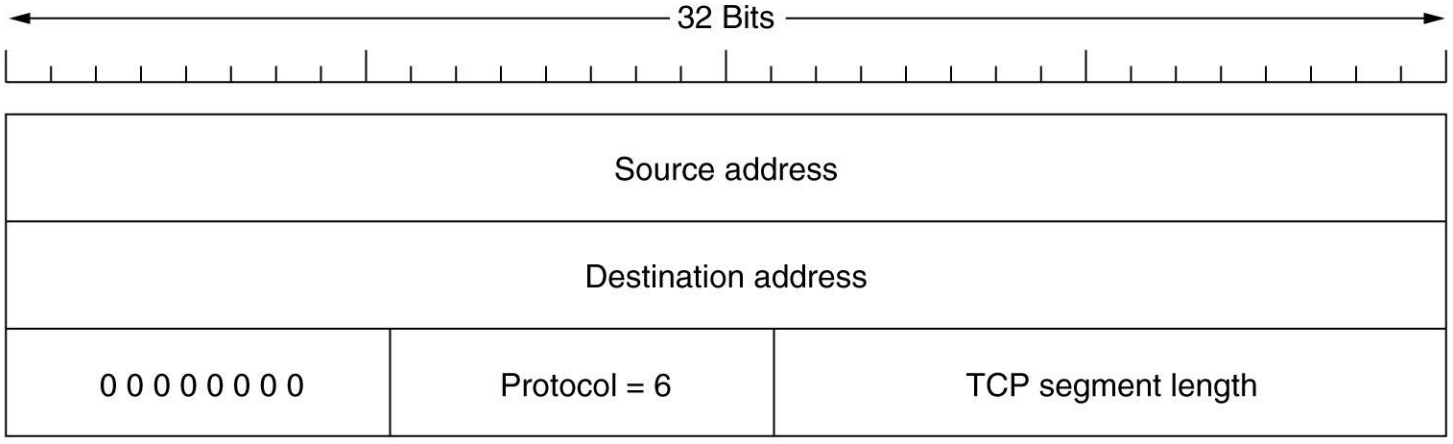
- 当紧急指针使用的时候，URG 被置为1。紧急指针是一个对于当前序列号的字节偏移量，标明紧急数据从哪里开始P459
  - 当URG=1时，表明有紧急数据，必须首先处理；
  - 与紧急指针配合使用；
  - 收方收到这样的数据后，马上处理，处理完后恢复正常操作；
  - 即使win=0，也可以发送这样的紧急数据段
- ACK 可设为 1/0
  - 1：表示确认号有效
  - 0：标明确认号无效

TCP数据段头(续) P430

- **PSH** 表示这是带有PUSH标志的数据
  - 接收方收到这样的数据，应该立刻送到上层，而不需要缓存它
- **RST** 被用来重置一个已经混乱的连接
- **SYN** 用在连接建立的过程
  - 当SYN=1，ACK=0， 连接请求
  - 当SYN=1，ACK=1， 连接接受
- **FIN** 被用来释放连接，它表示发送方已经没有数据要传输了，但是可以继续接收数据.

TCP数据段头(续) P430

- TCP中的流控(**Flow control**)使用一个可变长的滑动窗口来完成的
- **Window size** – 告诉对方可以发送的数据字节数（从确认字节号开始（决定于接收方）
- **Checksum** –提供额外的可靠性
  - 校验的范围包括头部、数据和概念性的伪头部



TCP数据段头(续) P431

- 选项域提供了一种增加基本头没有包含内容的方法
- 选项实例
  1. 最重要的选项是允许每台主机指定他愿意接收的最大TCP净荷长度
     - 使用大的数据段比使用小的数据段更高效
     - 在连接建立阶段，每方可以在选项中宣布他的最大TCP净荷长度，并查看对方的给出的最大值；选择双方中宣布小的那个使用
     - 缺省的值为 536 bytes，所有互联网主机默认为可以接受536 + 20 = 556 bytes的数据段

TCP数据段头(续)

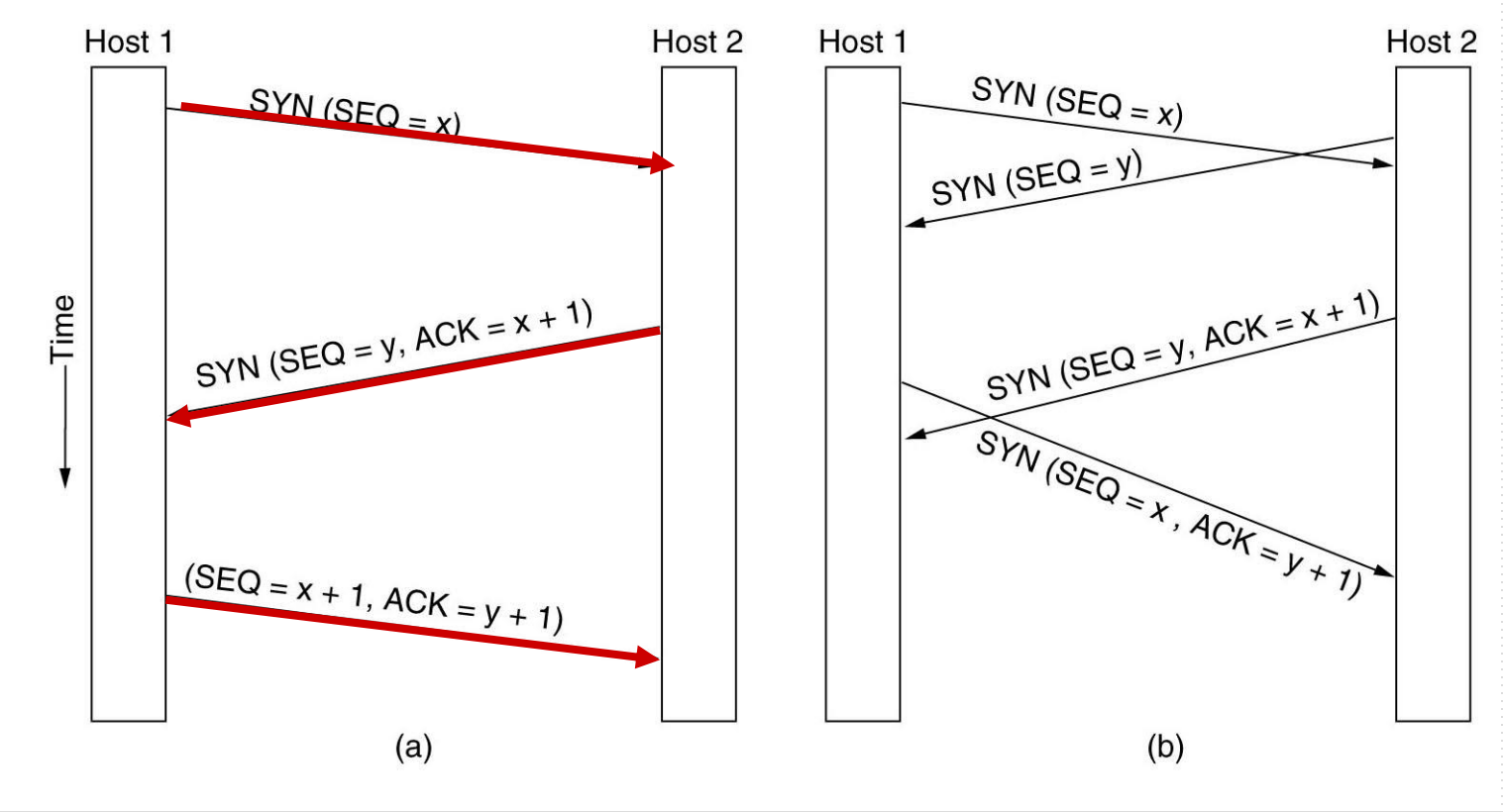
- 选项实例

2. 对于高带宽、高延迟或两者兼备的线路，64kB窗口可能是一个问题；窗口尺度（Window scale）选项允许收发双方协商一个窗口尺度因子， 这个因子允许双方把窗口尺寸域向左移动至14位，因此窗口数可多达 2^{30} 字节，很多TCP都支持这个选项
3. RFC 1106中描述的另一个选项，现在广泛实现了，即使用选择性重传（ selective repeat ），而不是回退n帧协议（ go back n ）

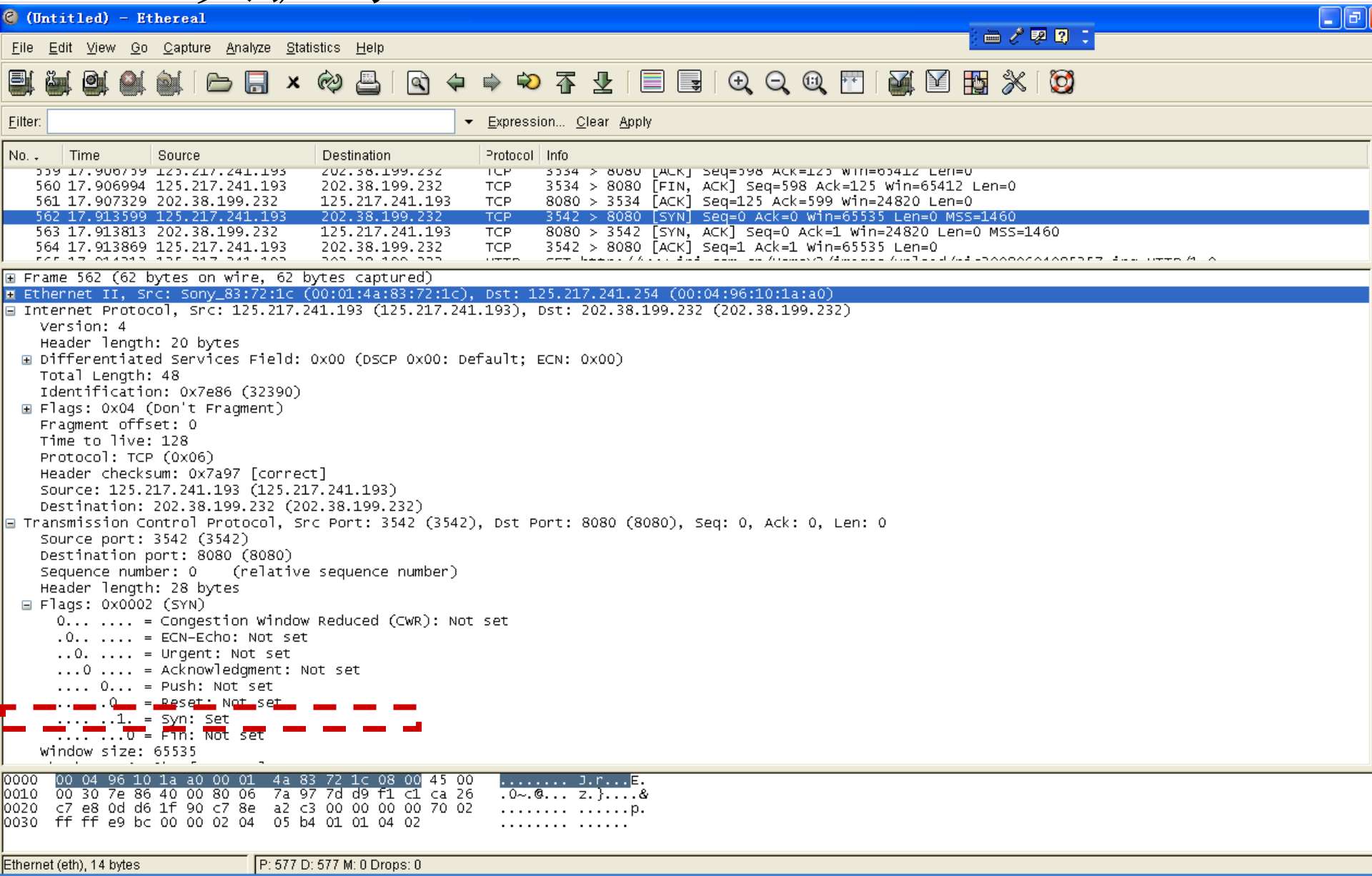
TCP 连接的建立（6.5.5 P432）

- 采用三次握手建立连接
  - 一方（server）被动地等待一个进来的连接请求
  - 另一方（the client）通过发送连接请求，设置一些参数
  - 服务器方回发确认应答，
  - 应答到达请求方，请求方最后确认，连接建立

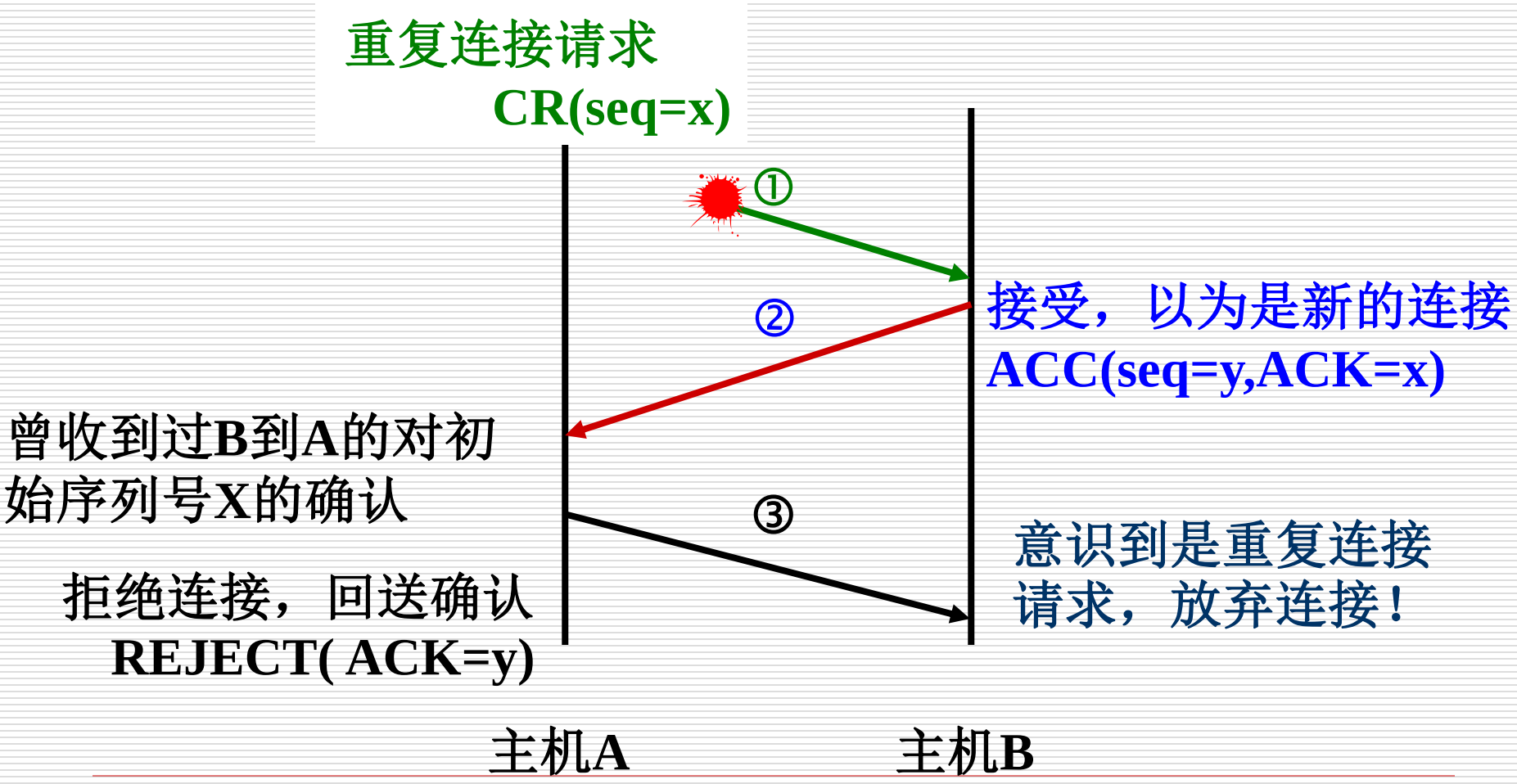
TCP 连接的建立（P432）



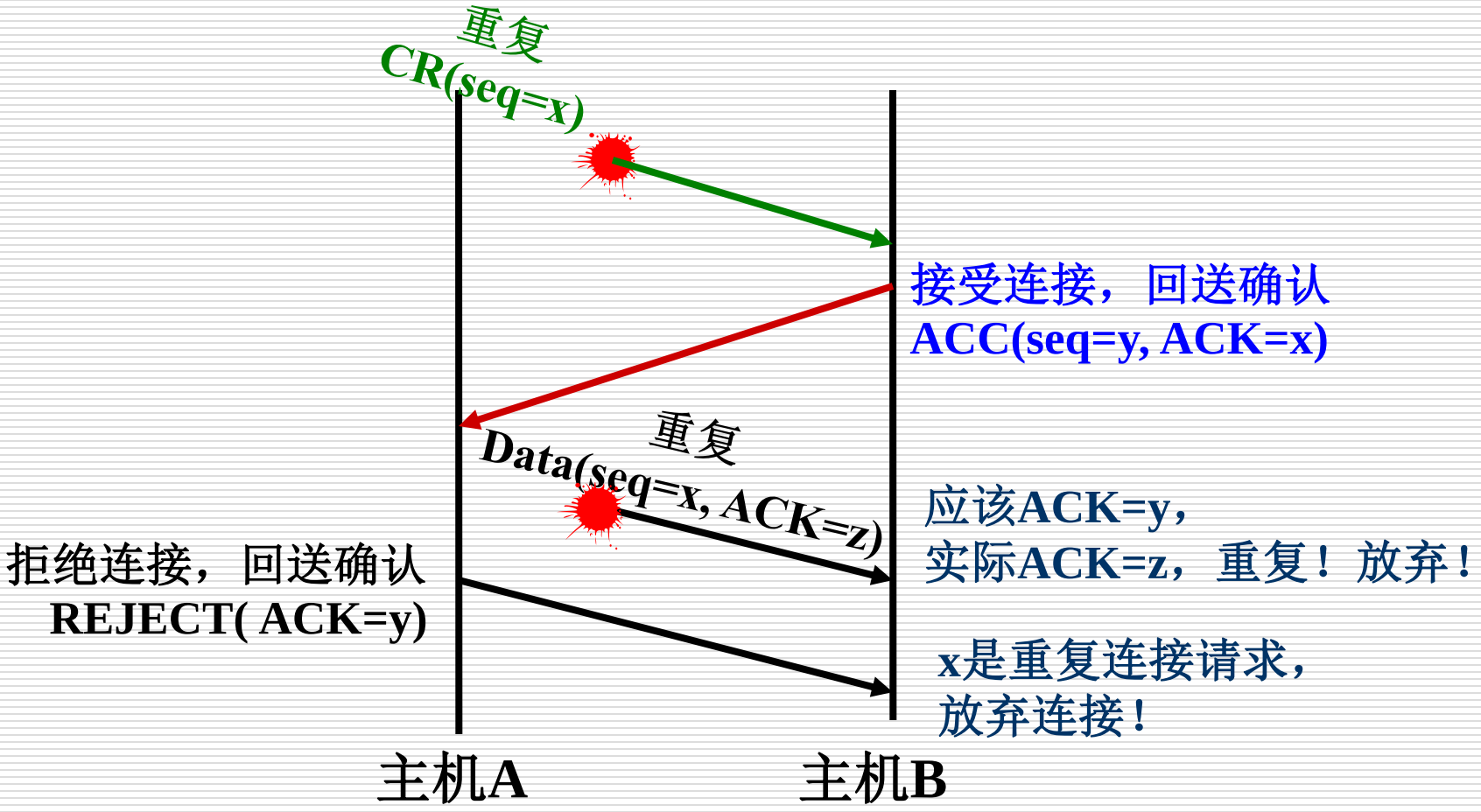
三次握手P432


(Untitled) - Ethereal
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
No. Time Source Destination Protocol Info
560 17.906994 125.217.241.193 202.38.199.232 TCP 3534 > 8080 [FIN, ACK] Seq=598 Ack=125 Win=65412 Len=0
561 17.907329 202.38.199.232 125.217.241.193 TCP 8080 > 3534 [ACK] Seq=125 Ack=599 Win=24820 Len=0
562 17.913599 125.217.241.193 202.38.199.232 TCP 3542 > 8080 [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
563 17.913813 202.38.199.232 125.217.241.193 TCP 8080 > 3542 [SYN, ACK] Seq=0 Ack=1 Win=24820 Len=0 MSS=1460
564 17.913869 125.217.241.193 202.38.199.232 TCP 3542 > 8080 [ACK] Seq=1 Ack=1 Win=65535 Len=0
Frame 562 (62 bytes on wire, 62 bytes captured)
Ethernet II, Src: Sony_83:72:1c (00:01:4a:83:72:1c), Dst: 125.217.241.254 (00:04:96:10:1a:a0)
Internet Protocol, Src: 125.217.241.193 (125.217.241.193), Dst: 202.38.199.232 (202.38.199.232)
Version: 4
Header length: 20 bytes
Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
Total Length: 48
Identification: 0x7e86 (32390)
Flags: 0x04 (Don't Fragment)
Fragment offset: 0
Time to live: 128
Protocol: TCP (0x06)
Header checksum: 0x7a97 [correct]
Source: 125.217.241.193 (125.217.241.193)
Destination: 202.38.199.232 (202.38.199.232)
Transmission Control Protocol, Src Port: 3542 (3542), Dst Port: 8080 (8080), Seq: 0, Ack: 0, Len: 0
Source port: 3542 (3542)
Destination port: 8080 (8080)
Sequence number: 0 (relative sequence number)
Header length: 28 bytes
Flags: 0x0002 (SYN)
0... .... = Congestion Window Reduced (CWR): Not set
.0.. .... = ECN-Echo: Not set
..0. .... = Urgent: Not set
...0 .... = Acknowledgment: Not set
.... 0... = Push: Not set
.... .0.. = Reset: Not set
.... ..1. = Syn: Set
....0 = Fin: Not set
Window size: 65535
0000 00 04 96 10 1a a0 00 01 4a 83 72 1c 08 00 45 00 .... J.r...E.
0010 00 30 7e 86 40 00 80 06 7a 97 7d d9 f1 c1 ca 26 .0~.@... z.}....&
0020 c7 e8 0d d6 1f 90 c7 8e a2 c3 00 00 00 00 70 02 ....p.
0030 ff ff e9 bc 00 00 02 04 05 b4 01 01 04 02 ....
Ethernet (eth), 14 bytes
P: 577 D: 577 M: 0 Drops: 0

重复连接请求CR


重复连接请求
CR(seq=x)
①
接受，以为是新的连接
②
ACC(seq=y,ACK=x)
曾收到过B到A的对初始序列号X的确认
③
意识到是重复连接请求，放弃连接！
拒绝连接，回送确认
REJECT(ACK=y)
主机A
主机B

重复CR与重复ACK

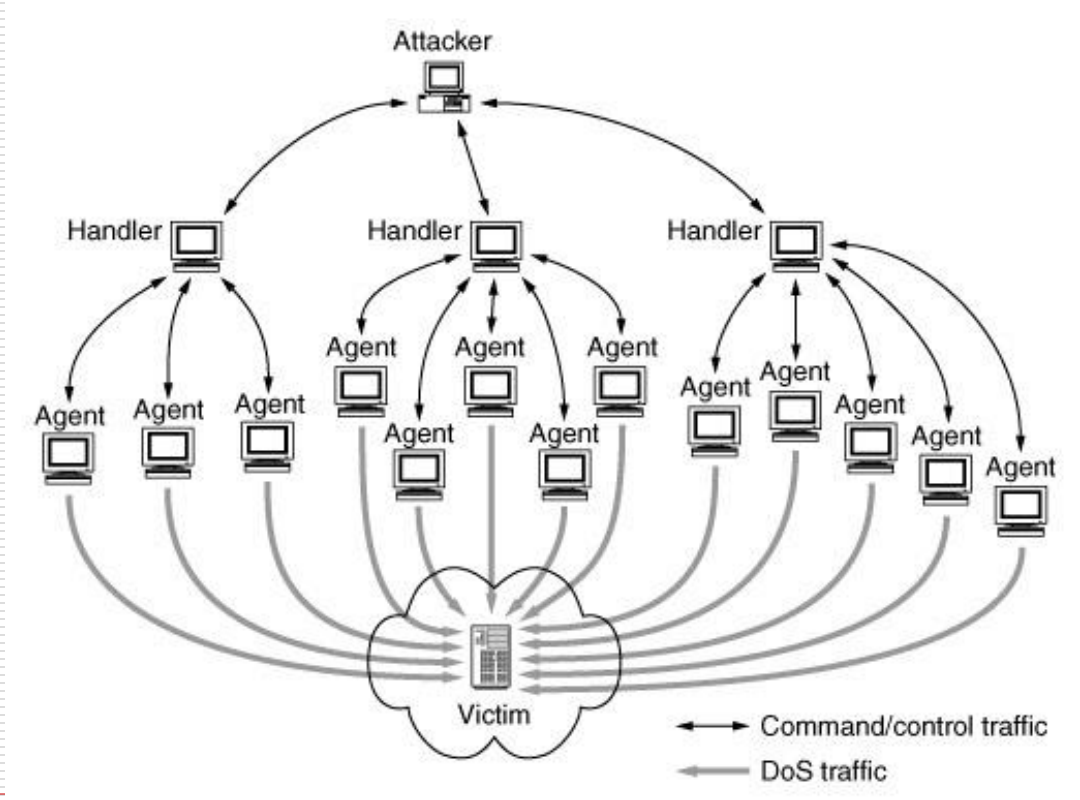

重复
CR(seq=x)
接受连接，回送确认
ACC(seq=y, ACK=x)
重复
Data(seq=x, ACK=z)
应该ACK=y，
实际ACK=z，重复！放弃！
拒绝连接，回送确认
REJECT(ACK=y)
x是重复连接请求，
放弃连接！
主机A
主机B

注意

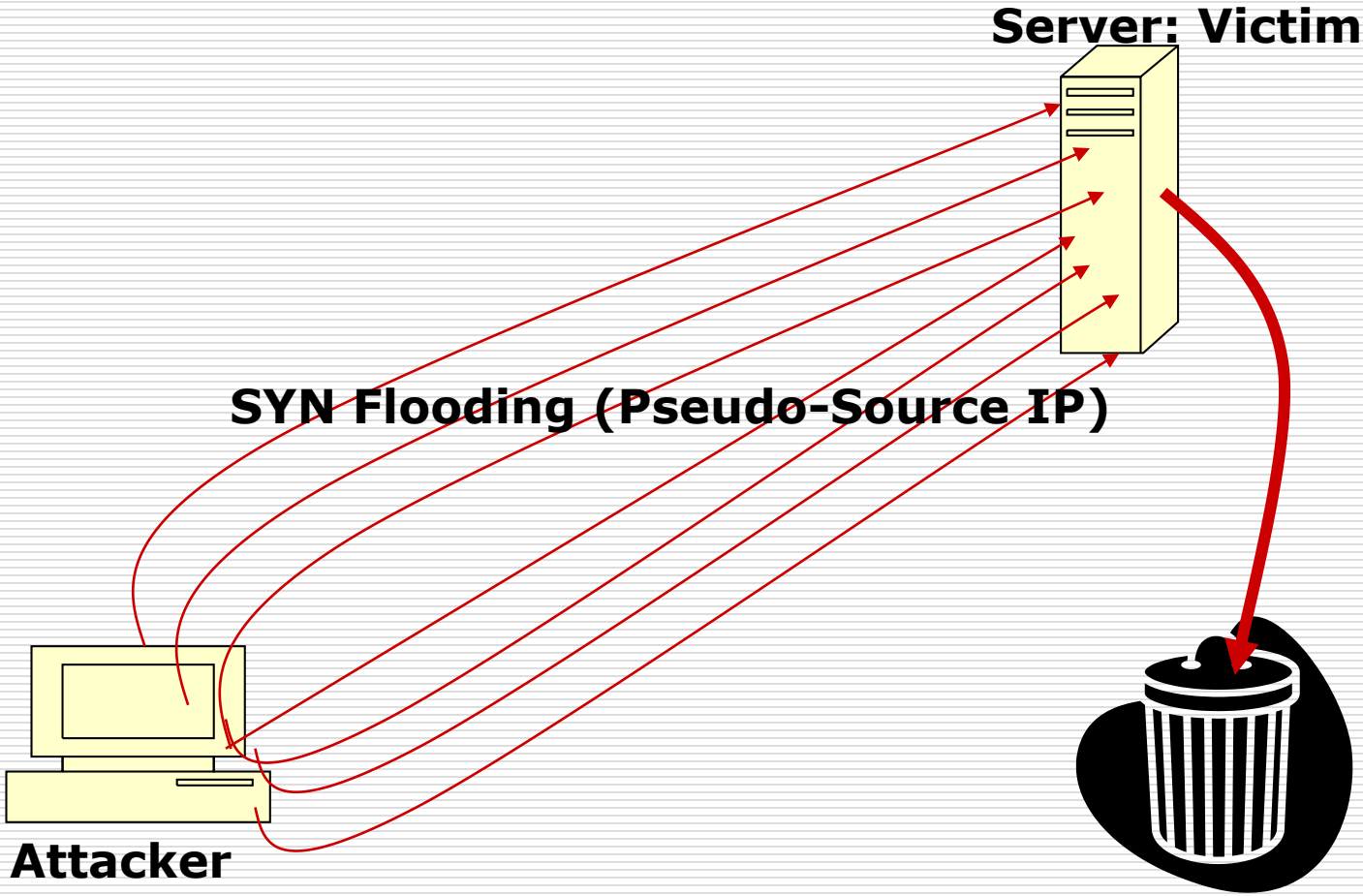
- SYN泛洪导致DoS攻击（伪造源IP）
- 数据传输开始后可能有两个原因导致阻塞
  - 快的机器向慢的机器发送数据
  - 多台机器同时向一台机器发送数据
- 拥塞避免方法：确认技术、窗口技术

拒绝服务攻击DoS

- **SYN Flooding can result in DoS (deny of service) attack**



图示：SYN Flooding



课堂练习 1

描述符

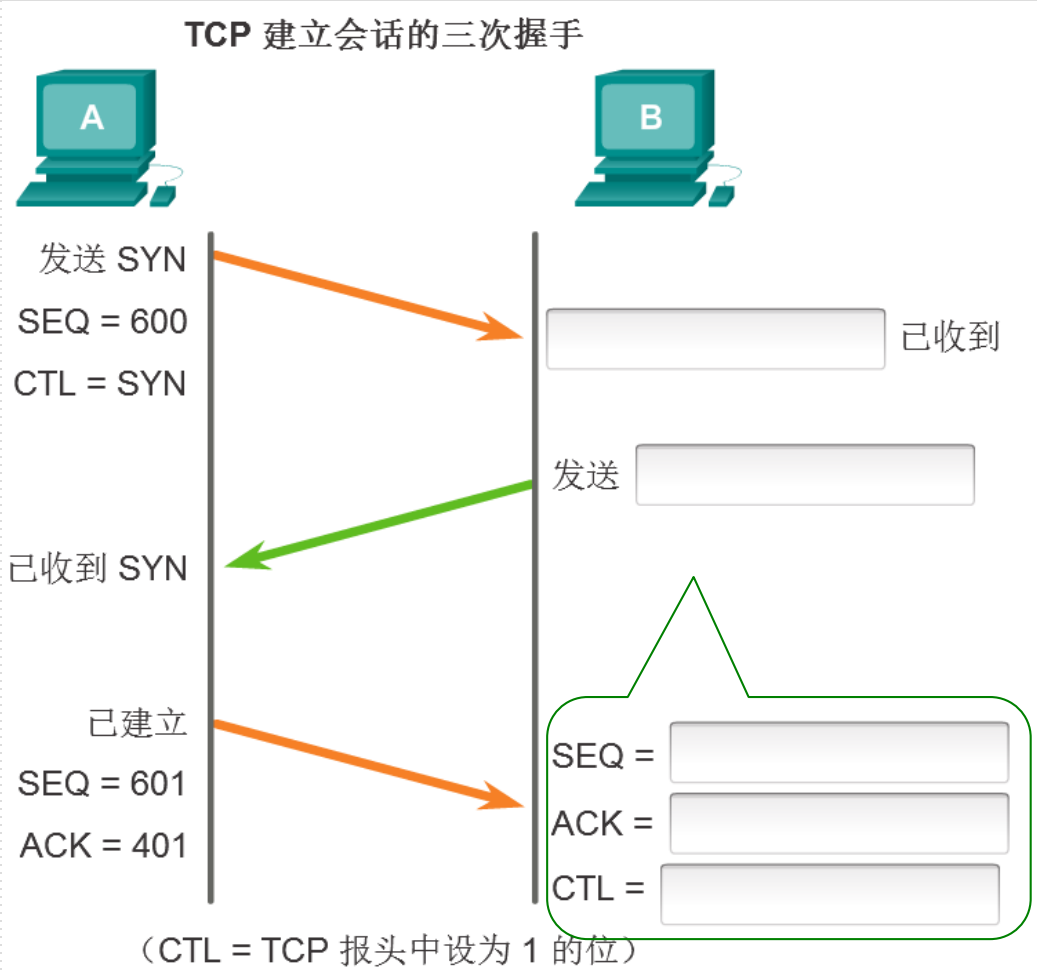
601

SYN、ACK

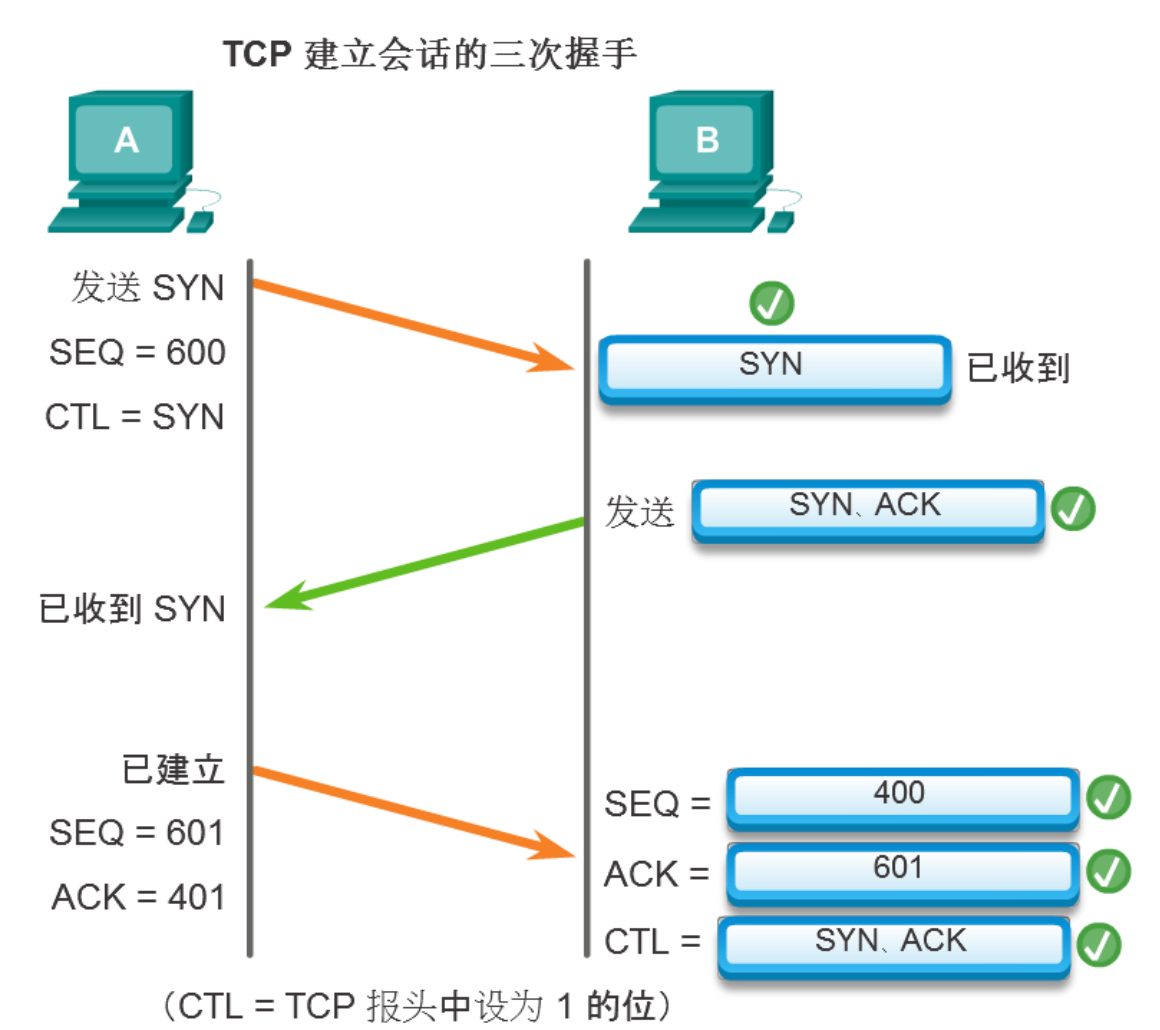
SYN、ACK

SYN

400



课堂练习1参考答案


TCP 建立会话的三次握手
A
B
发送 SYN
SEQ = 600
CTL = SYN
SYN
已收到
发送
SYN、ACK
已收到 SYN
已建立
SEQ = 601
ACK = 401
SEQ =
400
ACK =
601
CTL =
SYN、ACK
（CTL = TCP 报头中设为 1 的位）

课堂练习2

- 主机 H 要访问服务器 S, **试填写三次握手过程的数据段头部。假设 H的 ISNs (初始序列号)和端口分别是 200 和3000，S的 ISNs 和端口分别是 500 和 80**

填写

SYN=?ACK=?

H向S请求

Source port=_____	**Destination port=_____**
Sequence number=_____	
Acknowledge number=_____	

S回应请求

Source port=_____	**Destination port=_____**
Sequence number=_____	
Acknowledge number=_____	

H最后确认

Source port=_____	**Destination port=_____**
Sequence number=_____	
Acknowledge number=_____	

参考答案

SYN=1 ACK=0

H向S请求

Source port=___3000_____	Destination port=__80_____
Sequence number=___200_____	
Acknowledge number=_____	

SYN=1 ACK=1

S回应请求

Source port=_____80_____	Destination port=____3000_____
Sequence number=___500_____	
Acknowledge number=____201_____	

SYN=0 ACK=1

H最后确认

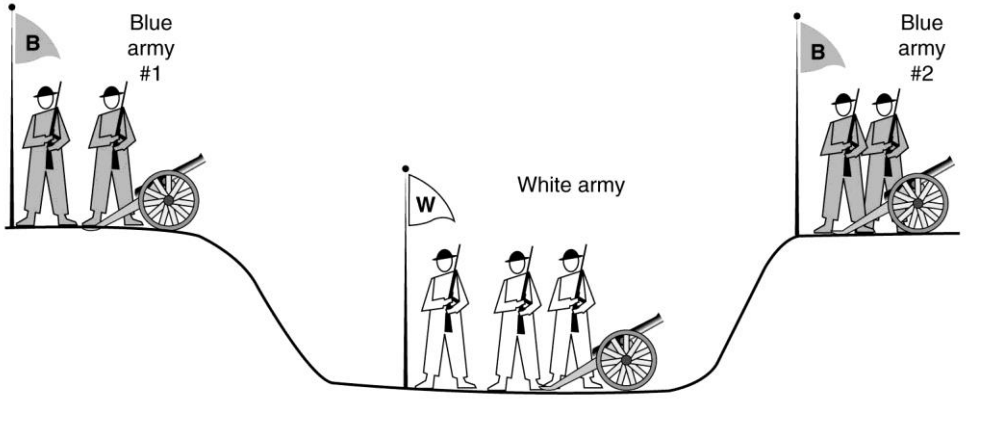
Source port=_____3000_____	Destination port=___80_____
Sequence number=__201_____	
Acknowledge number=_____501_____	

TCP 连接释放（P433、P400）

- 释放连接
  - 任何一方在没有数据要传送的时候，都可以发送一个FIN置位了的 TCP 数据段
  - 当FIN被确认的时候，该方向的连接被关闭
  - 当双向连接都关闭了的时候，连接释放
- 为了避免两军队（two-army）问题，使用定时器
  - 如果一方发送了FIN数据段出去却在一个设定的时间没有收到应答，释放连接
  - 另一方最终会注意到连接的对方已经不在了，超时后连接释放

两军队问题P400

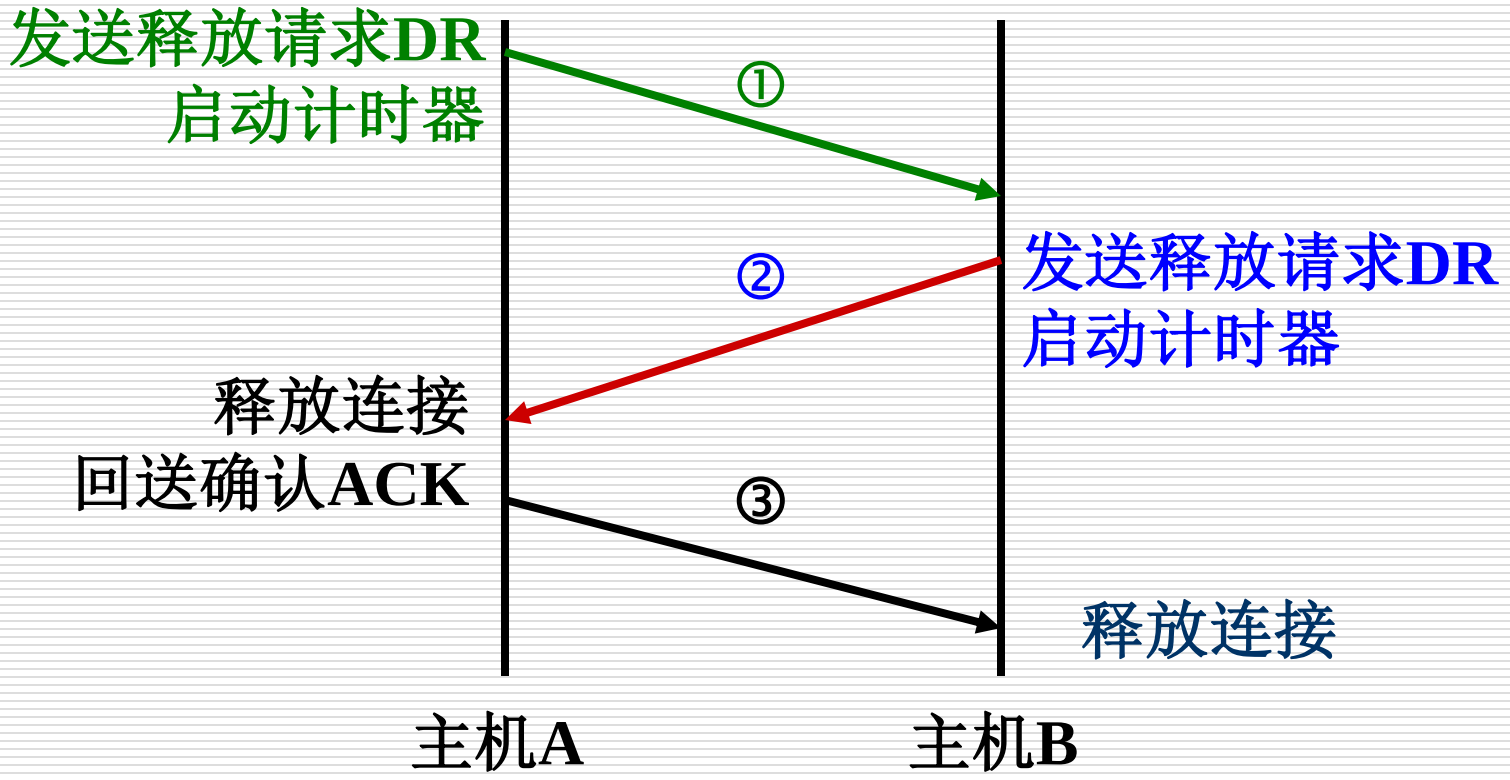
- 对称释放–对待连接像两个双向连接，要求连接的两端都释放
- 遗憾的是，决定什么时候两边释放非常困难
- 这个难题被称为两军队问题（**two-army problem**）
  - 可以证明，解决这一难题的协议并不存在P400
  - 最后信息的发送者，永远无法知道这个信息是否到达



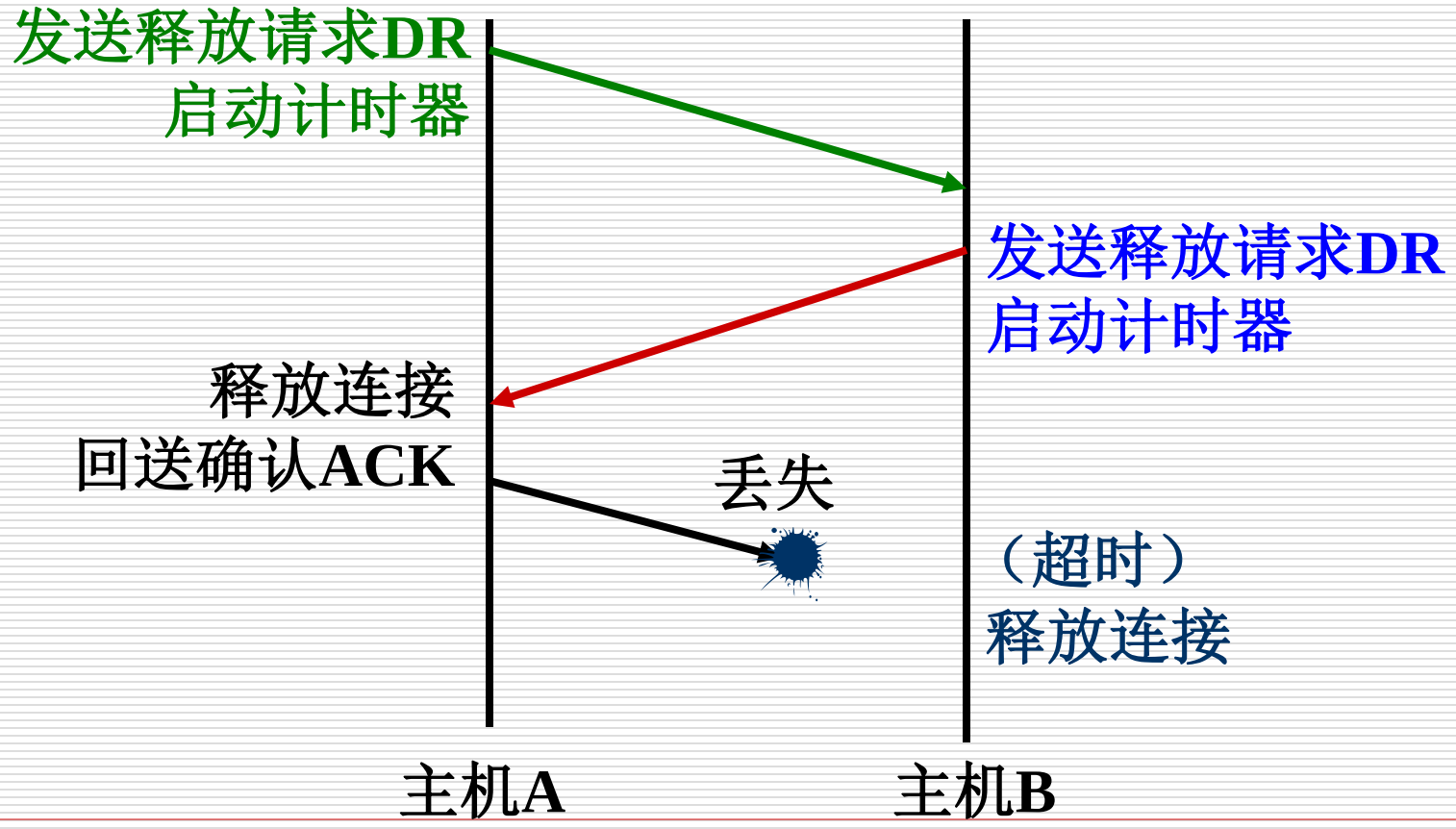
连接释放（续）

- 理论上讲，如果初始DR的和重传都丢了，协议失败
  - 发送者将放弃发送且释放连接，但是，另外一端却不知道这些情况，仍然处于活跃的状态
  - 这种情形导致半开放连接（**half- open**）
- 杀死半开放连接的方式P401
  - 如果在一定的时间内，没有TPDUs 到达的话，连接自动释放
  - 如果这样，传输实体在发送一个TPDU的时候必须启动定时器，定时器超期，将发动一个哑TPDU（dummy TPDU），以免被断掉 P430

三次握手正常释放连接P402



最后的确认TPDU丢失



课堂练习

将每个描述符拖动到演示 TCP 建立 会话 4 步流程的图像中的相应位置。

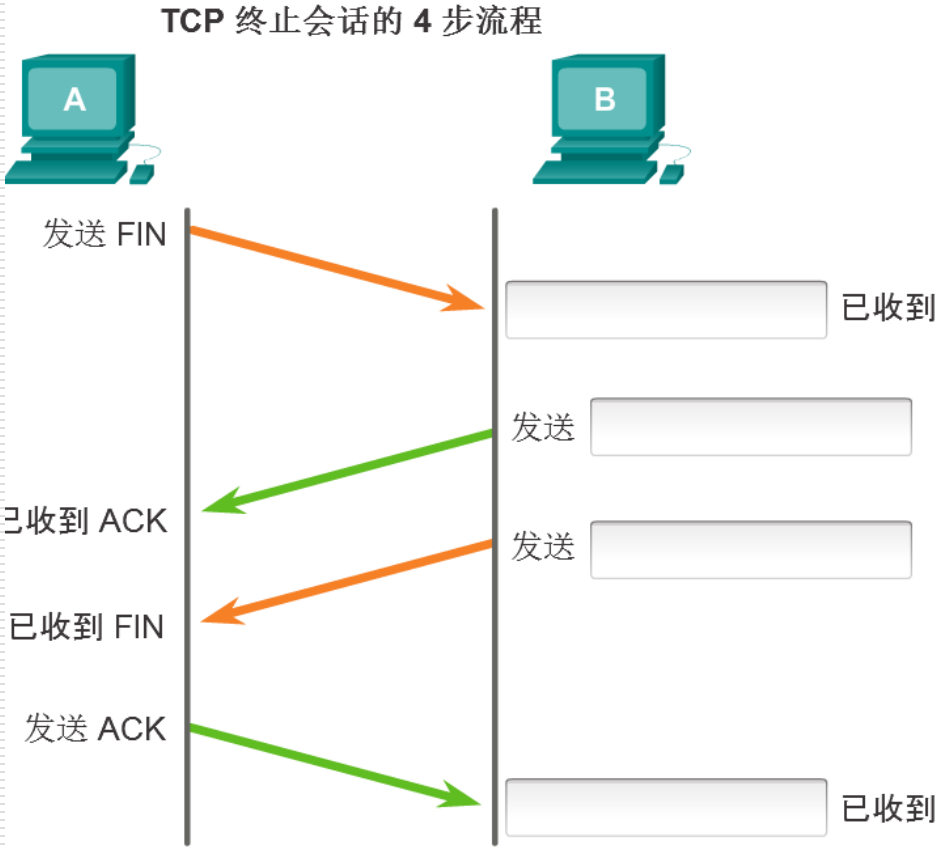
描述符

FIN

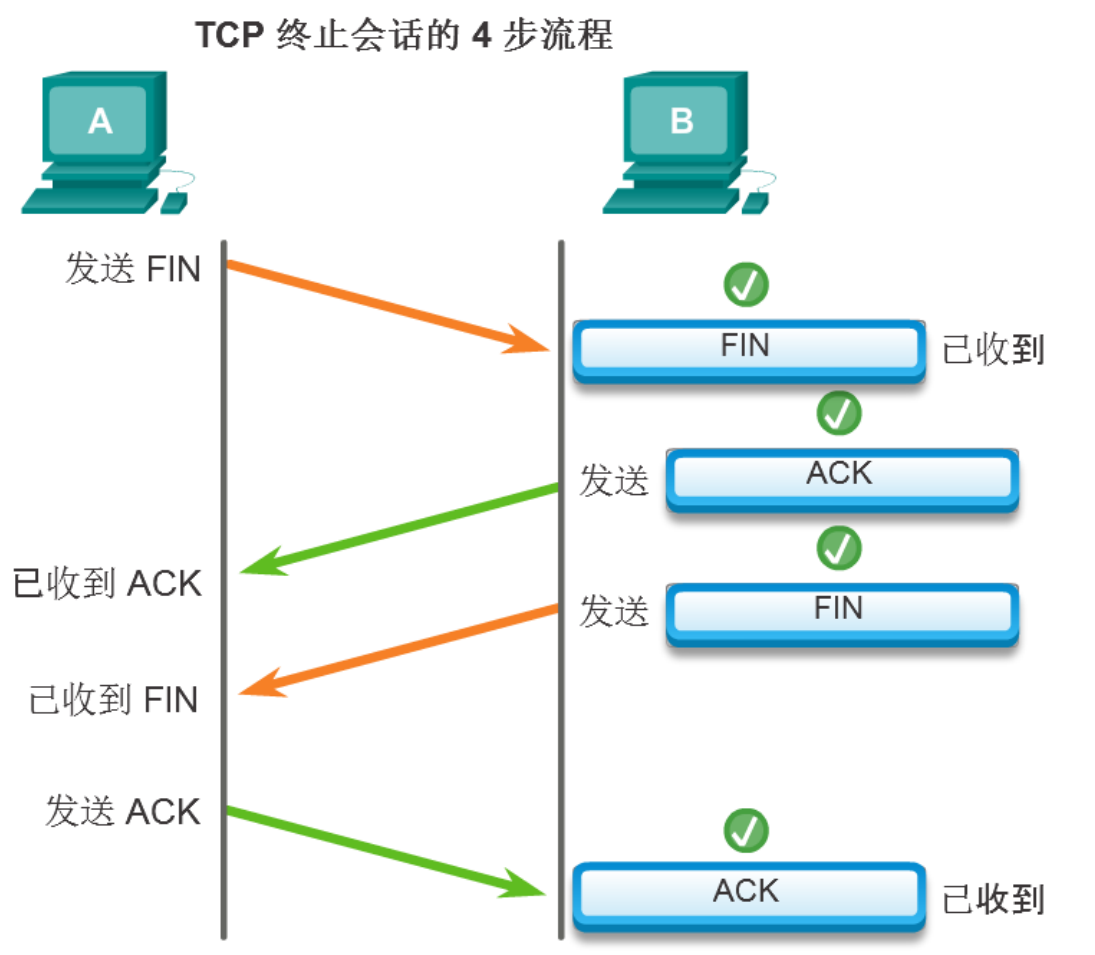
FIN

ACK

ACK

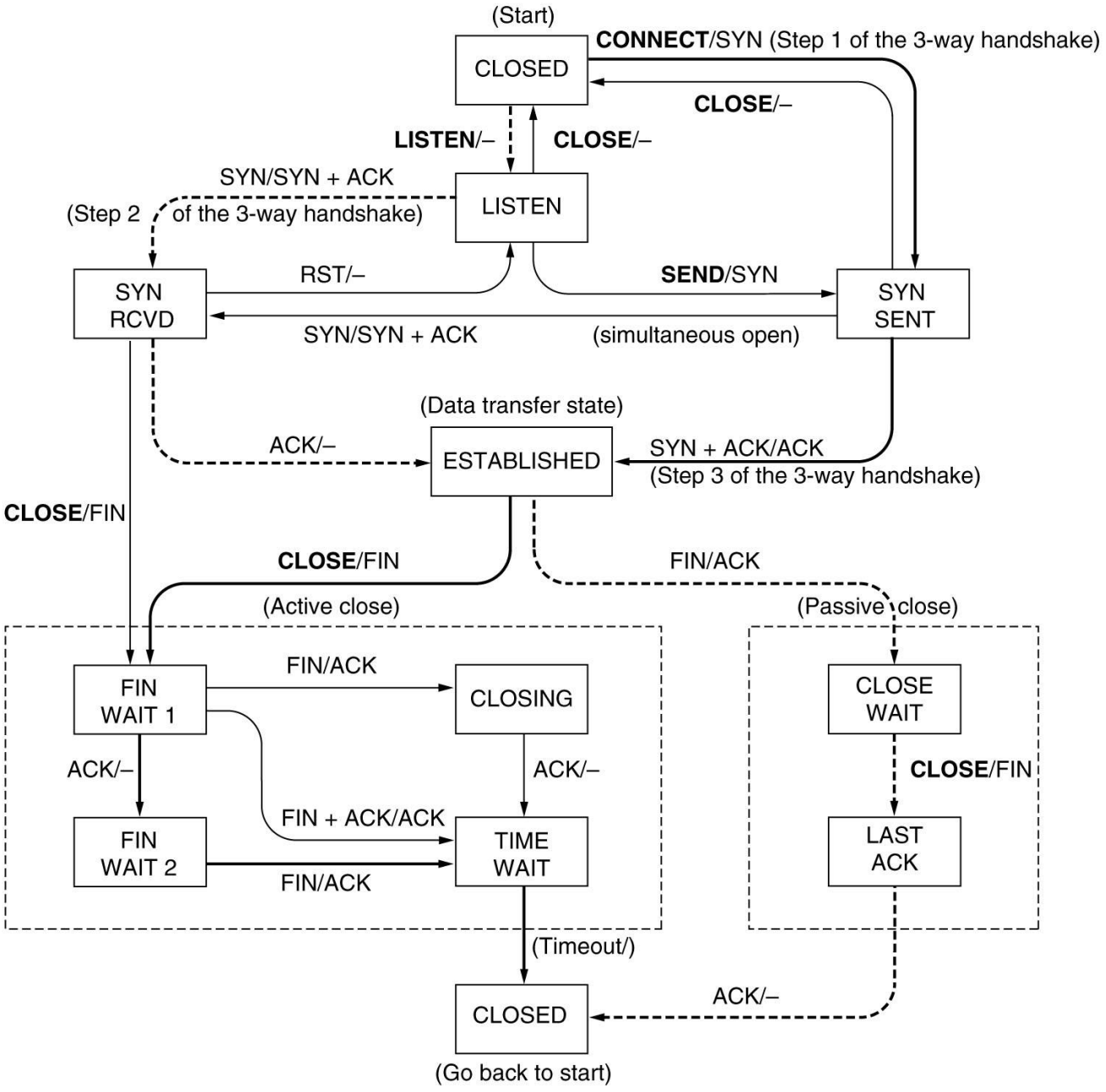


参考答案

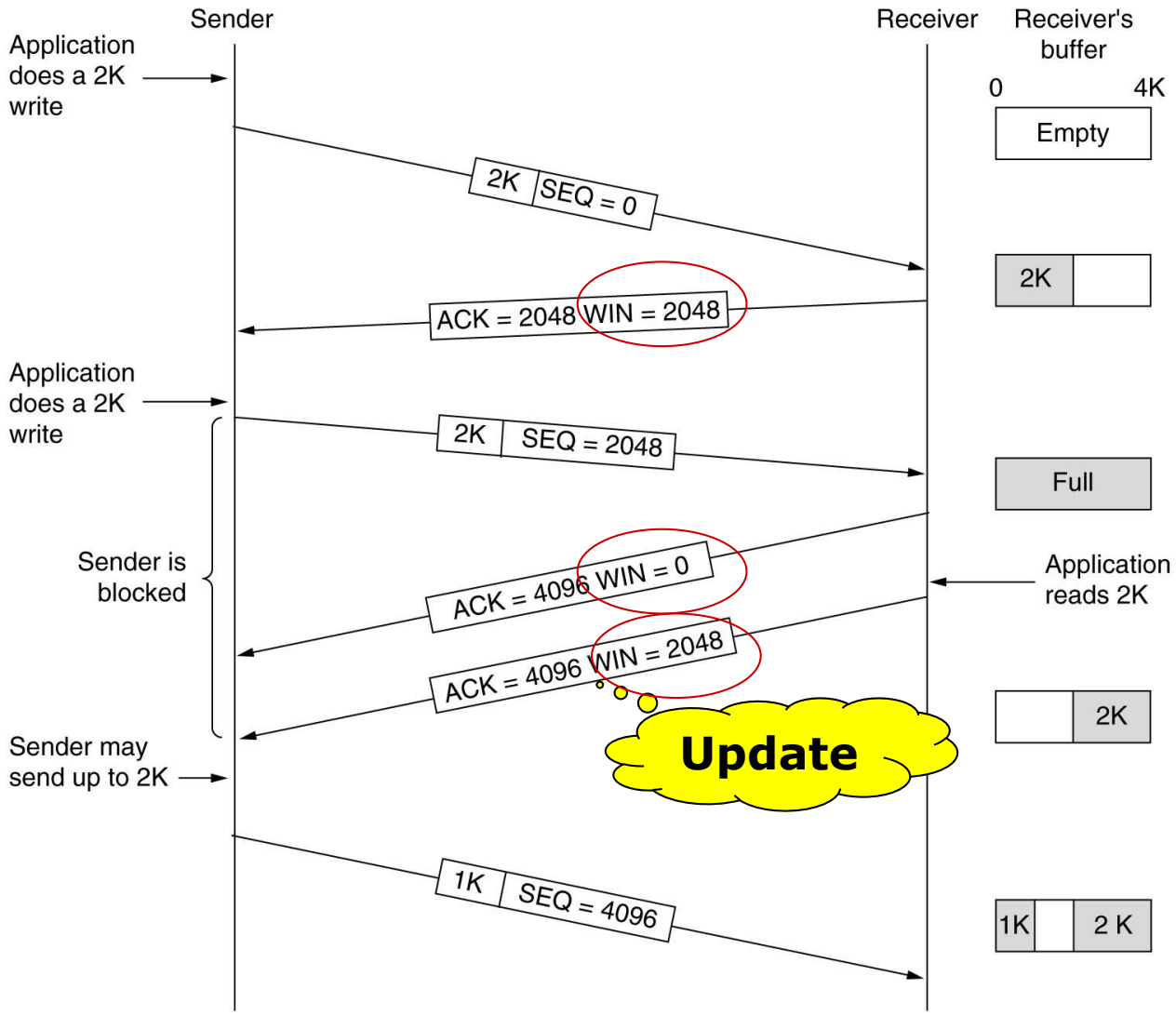

TCP 终止会话的 4 步流程
A
B
发送 FIN
FIN
已收到
发送
ACK
已收到 ACK
发送
FIN
已收到 FIN
发送 ACK
ACK
已收到

TCP 连接管理模型 P434

State	Description
CLOSED	No connection is active or pending
LISTEN	The server is waiting for an incoming call
SYN RCVD	A connection request has arrived; wait for ACK
SYN SENT	The application has started to open a connection
ESTABLISHED	The normal data transfer state
FIN WAIT 1	The application has said it is finished
FIN WAIT 2	The other side has agreed to release
TIMED WAIT	Wait for all packets to die off
CLOSING	Both sides have tried to close simultaneously
CLOSE WAIT	The other side has initiated a release
LAST ACK	Wait for all packets to die off


(Start)
CLOSED
CONNECT/SYN (Step 1 of the 3-way handshake)
CLOSE/–
LISTEN/–
CLOSE/–
SYN/SYN + ACK
(Step 2 of the 3-way handshake)
LISTEN
SYN RCVD
RST/–
SEND/SYN
SYN SENT
SYN/SYN + ACK
(simultaneous open)
(Data transfer state)
ACK/–
ESTABLISHED
SYN + ACK/ACK
(Step 3 of the 3-way handshake)
CLOSE/FIN
CLOSE/FIN
FIN/ACK
(Active close)
(Passive close)
FIN WAIT 1
FIN/ACK
CLOSING
CLOSE WAIT
ACK/–
ACK/–
CLOSE/FIN
FIN + ACK/ACK
FIN WAIT 2
TIME WAIT
LAST ACK
FIN/ACK
(Timeout/)
ACK/–
CLOSED
(Go back to start)

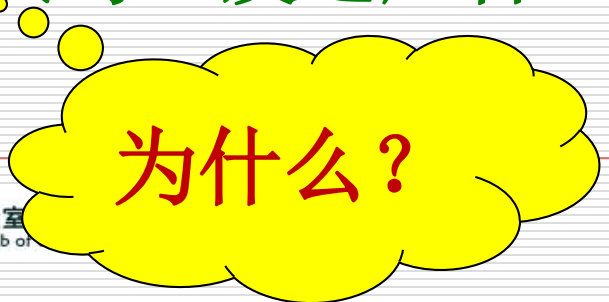
TCP 传输策略P436



窗口尺寸受制于接收方！

TCP 传输层策略（续）

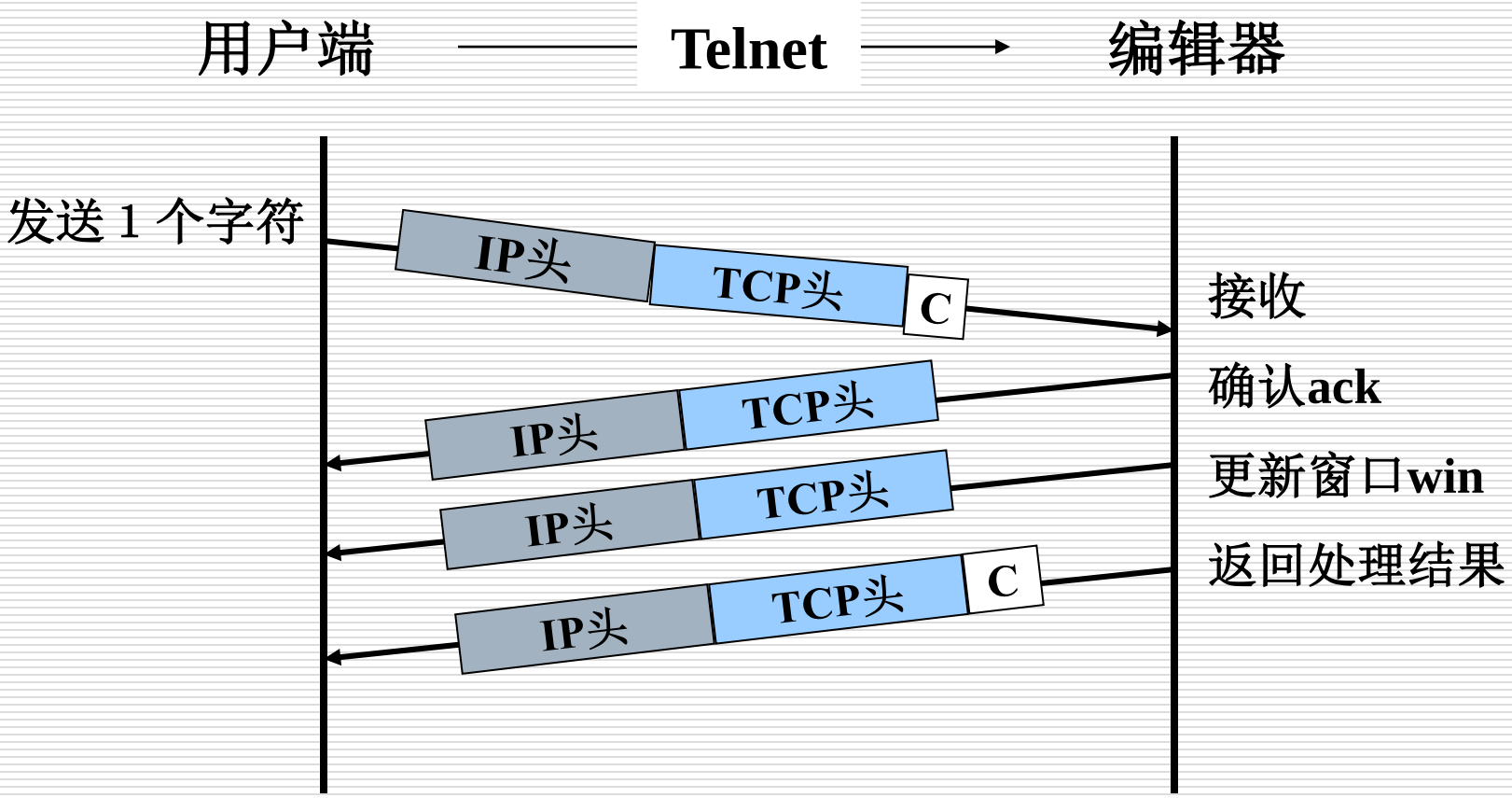
- 当窗口数为 0 时，发送者不能正常发送数据段，除非：
  - Urgent数据。比如，用户想杀掉远端机器上的进程的时候，可以发送数据
  - 发送者可以发送一个字节的数据段，以便让接收者再次发送期待接收的字节号和窗口数（避免死锁）
- 发送者不需要马上发送应用程序产生的数据
- 接收者也不需要马上发送应答（当收到数据的时候）



TCP 传输策略 (P436~438)

- 考虑一个 指向某交互式编辑器（远程）的TELNET 连接，该编辑器对用户的每次击键都作出响应，在最坏的情况下：
  - 当用户敲入一个字符的时候，被送到传输实体，创建一个21字节的数据段，在传到网络层，变成了41字节的IP分组
  - 接收方（运行着编辑器的远端机）收到这个信息后，会立发送一个40字节的确认分组（20字节的 TCP段头和20字节的IP头）
  - 随后，当编辑器读取出这个字节，TCP实体发送一个窗口更新，这个分组也是40字节
  - 最后，当编辑器处理了这个字符，它发送一个41字节的分组作为该字符的回显
- 总共累计起来，对于每个敲入的字符，需要至少 162 字节的带宽（还没有考虑到链路层的开销），发送4个数据段。

远程交互telnet的最坏情形图示

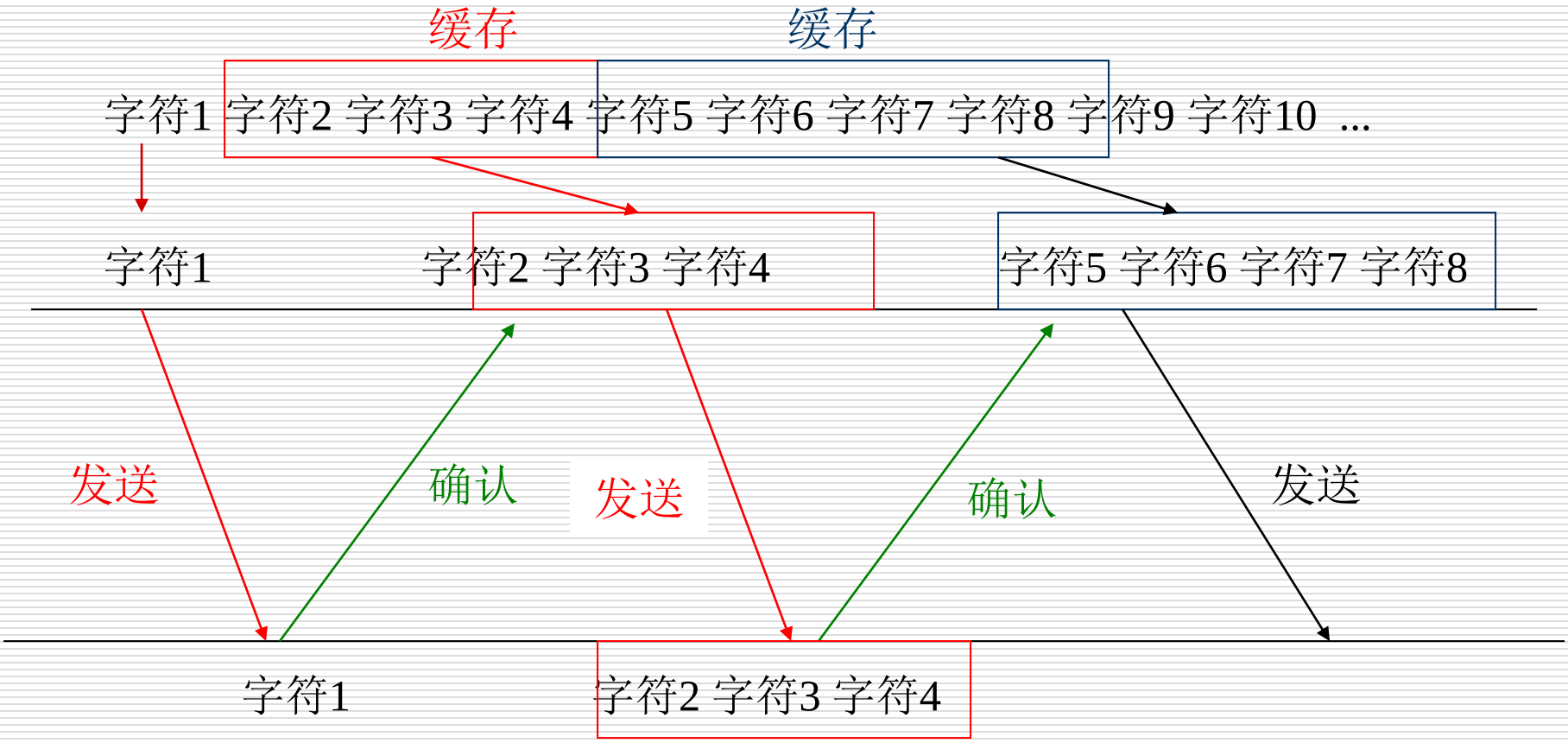


处理一个字符需要传送4次数据段，占用162字节带宽

TCP 传输策略 (P437)

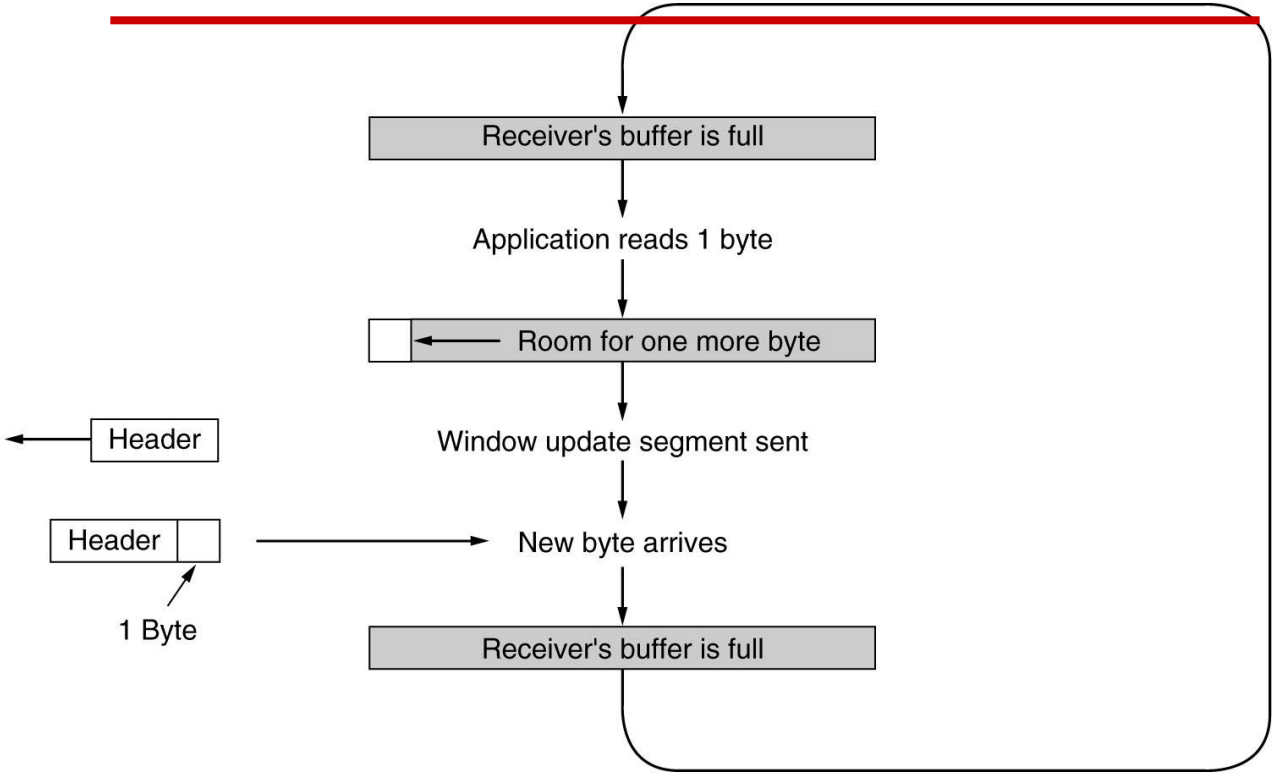
- 怎样优化接收端?
  - 接收端可以推迟500ms发送确认分组和窗口更新窗口，以便可以免费搭载在处理后的回显分组内（free ride）
- 怎样优化发送端 ？
  - Nagle's algorithm (1984):
    - 当数据以一次一字节的速度到达的时候，只发送第一个字节，然后将后续的字节缓存起来，直到发出的字节得到确认
    - 然后将缓存起来的字节在一个数据段中发出，再继续缓存，直到发出的数据得到确认
    - Nagle算法在很多TCP上实现，但是有些时候最好禁用，比如：当一个X-Windows应用在互联网运行的时候，鼠标的移动事件必须发送给远程计算机，把这些移动事件收集起来一批一批发送出去，使得鼠标的移动极不连贯

Nagle's 算法图示P437

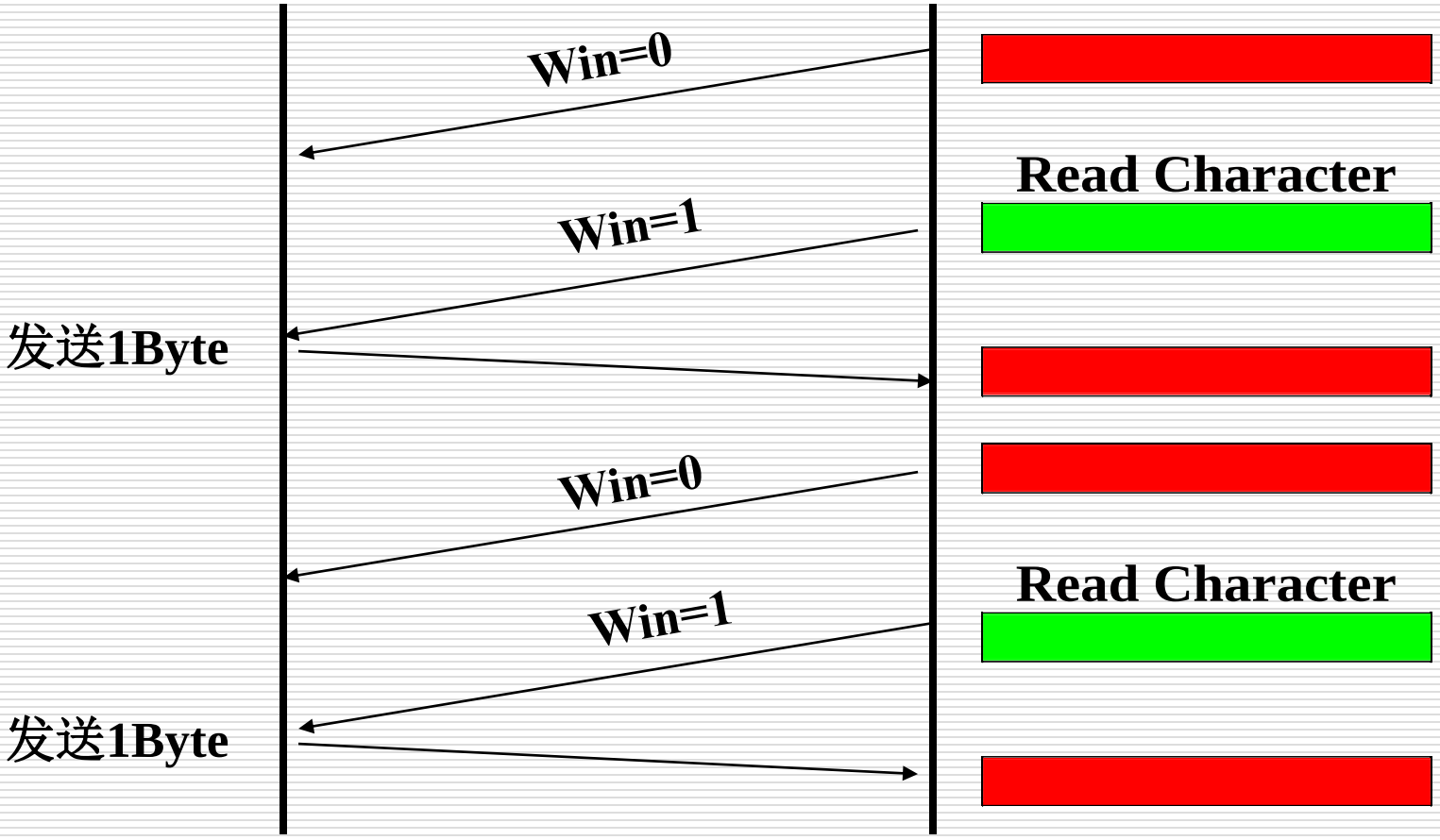


傻瓜窗口综合症 P437

- 另一个使TCP性能退化的问题是傻瓜窗口综合症（**silly window syndrome problem**）：当有大块数据被传递给发送端TCP实体，但接收端的交互式应用每次只读取一个字节的时候，问题就来了



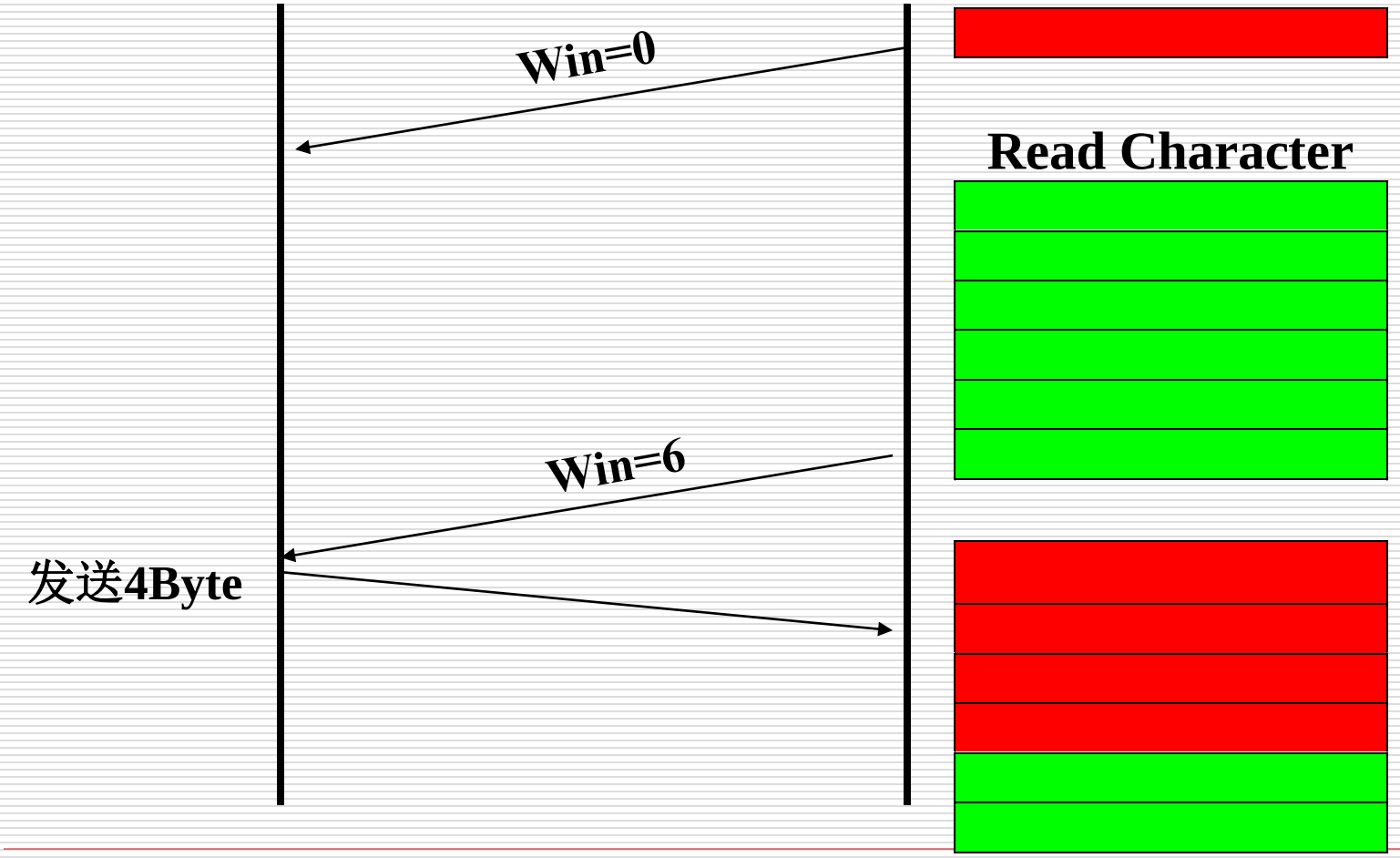
每次接收1字节


Win=0
Read Character
Win=1
发送1Byte
Win=0
Read Character
Win=1
发送1Byte

傻瓜窗口综合症 (续)

- **Clark解决方案**：阻止接收方发送只有1个字节的窗口更新，相反，它必须等待一段时间，当有了一定数量的空间之后再告诉发送方
- 而且，发送方不发送太小的数据段，相反，他试着等待一段时间，直到积累足够的窗口空间以便发送一个满的数据段，或者至少包含接收方缓冲区的一半大小
- 接收方可以可以维护一个内部缓冲，且阻塞上层应用的 READ 请求，直到它有大块的数据提供

Clark解决方案



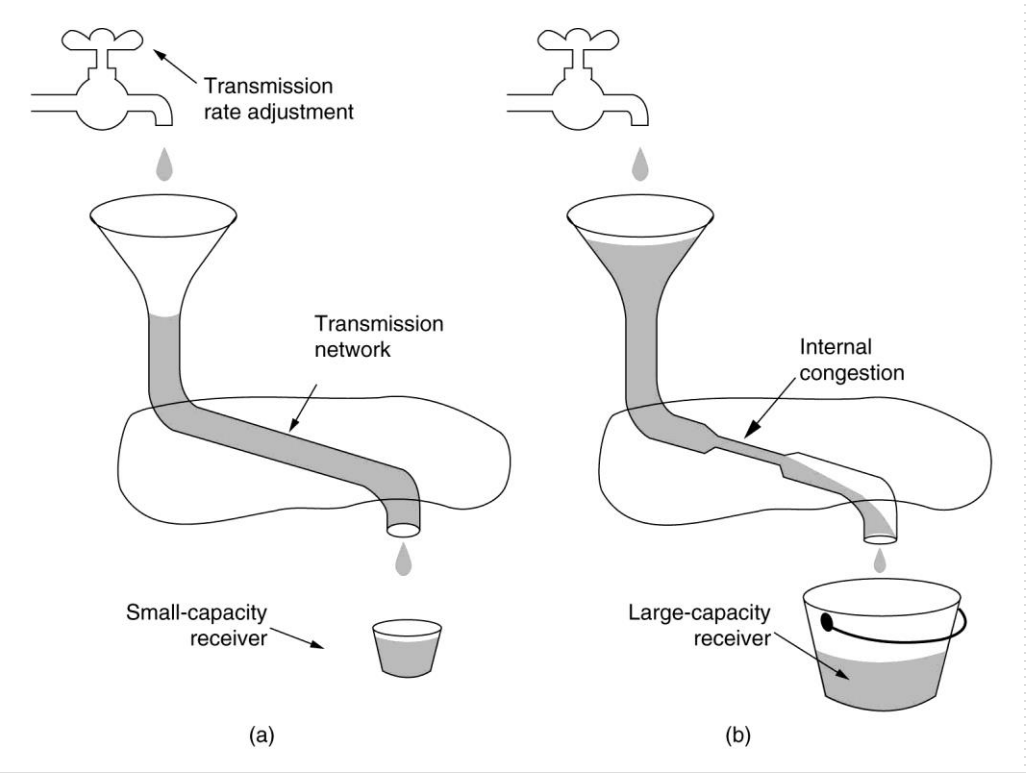
发送方和接收方

- 发送方（Nagle’s algorithm）
  - 尽量不发送数据含量小的数据段
  - 缓存应用层的数据，达到一定量再发送
- 接收方（Clark’s solution）
  - 不请求对方发送短数据段(window size)
  - 延迟窗口变更信息，使接收缓冲区足够大

TCP拥塞控制 P440

- 虽然网络层也试图管理拥塞，但是，大多数繁重的任务是由TCP来完成的，因为针对拥塞的真正解决方案是减慢数据率
- 分组守恒：当有一个老的分组离开之后才允许新的分组注入网络
- TCP希望通过动态维护窗口大小来实现这个目标
- **拥塞检测Congestion detection**
  - 所有的互联网TCP算法都假定超时是由拥塞引起的，并且通过监视超时的情况来判断是否出现问题
- **拥塞控制Congestion prevention**
  - 当一个连接建立的时候，双方选择一个合适的窗口大小，接收方根据自己的缓冲区大小来指定窗口的大小。
  - 如果发送者遵守此窗口大小的限制，则接收端不会出现缓冲区溢出的问题，但可能由于网络内部的拥塞而发生问题

TCP 拥塞控制 (续)



(a)快速的网络向小容量的接收方传输数据

(b)慢速的网络向大容量的接受方传输数据

TCP 拥塞控制 (P443)

- 互联网解决方案应该是认识到两个潜在的问题的：网络容量，接收者容量，然后单独地处理这两个问题
- 为此，每个发送者维护两个窗口：
  - 接收者窗口大小反映了目前窗口的容量 （容易控制）
  - 拥塞窗口大小反映了网络目前的容量（难于控制）
  - 发送者发送的数据字节数是两个窗口中小的那个窗口数

决定拥塞窗口的大小P443

- 慢启动算法（**Slow Start**） **(尝试的过程)：**
  - 当连接建立的时候，发送者用当前使用的最大数据段长度初始化拥塞窗口，然后发送一个最大的数据段
  - 如果在定时器超期之前收到确认，则将拥塞窗口翻倍，然后发送两个数据段。。。。。直至超时（或达到接收方窗口的大小）
  - 确定出拥塞窗口的大小
    - 如：如果试图发送 **4096** 字节没有问题，但是发送**8192**字节的时候，超时没有收到应答，则拥塞窗口设为**4096**个字节

慢速启动算法P443

按指数增长趋势定义拥塞窗口大小cwnd

- 初始：cwnd0 = MaxSegL（当前数据段长度）
- 增长：cwnd1 = 2 cwnd0

  cwnd2 = 2 cwnd1

  • • •
- 截止：达到接收窗口大小或超时

慢速启动算法图例

拥塞窗口二进制指数增长至接收窗口大小或超时

TCP 拥塞控制（P443）

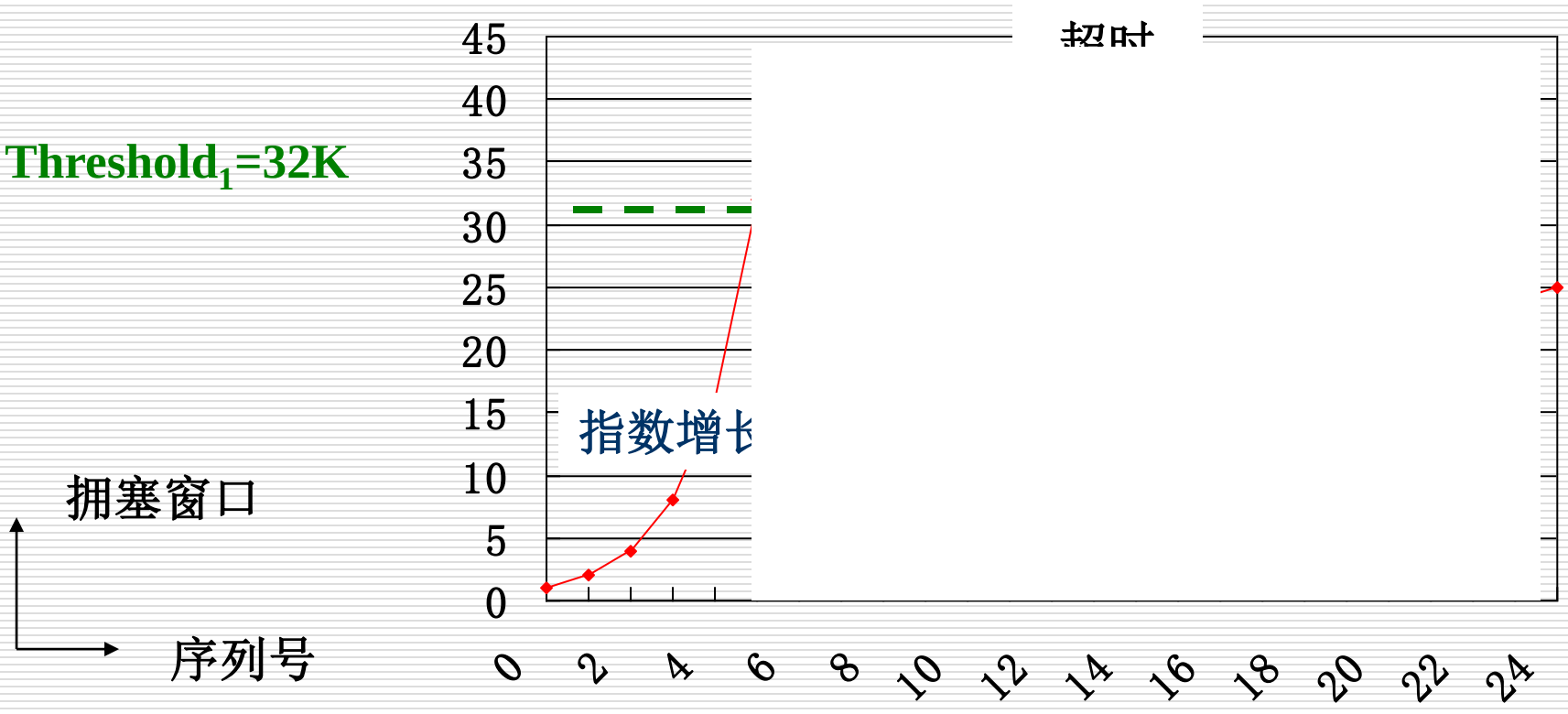
- 除了使用接收者窗口和拥塞窗口，TCP拥塞控制还是用了第三个参数，阈值（threshold），初始化为64K
- 当一个超时发生的时候，阈值降为当前拥塞窗口的一半，同时将拥塞窗口设为一个最大数据段的长度
- 然后使用慢启动算法来决定网络的容量，拥塞窗口增长到阈值时停止指数增长
- 从这个点开始，每次成功的传输都会让拥塞窗口线性增长（即每次仅增长一个最大的数据段长度）

拥塞控制算法-CWin指数增长

$Threshold_0$=64K

超时

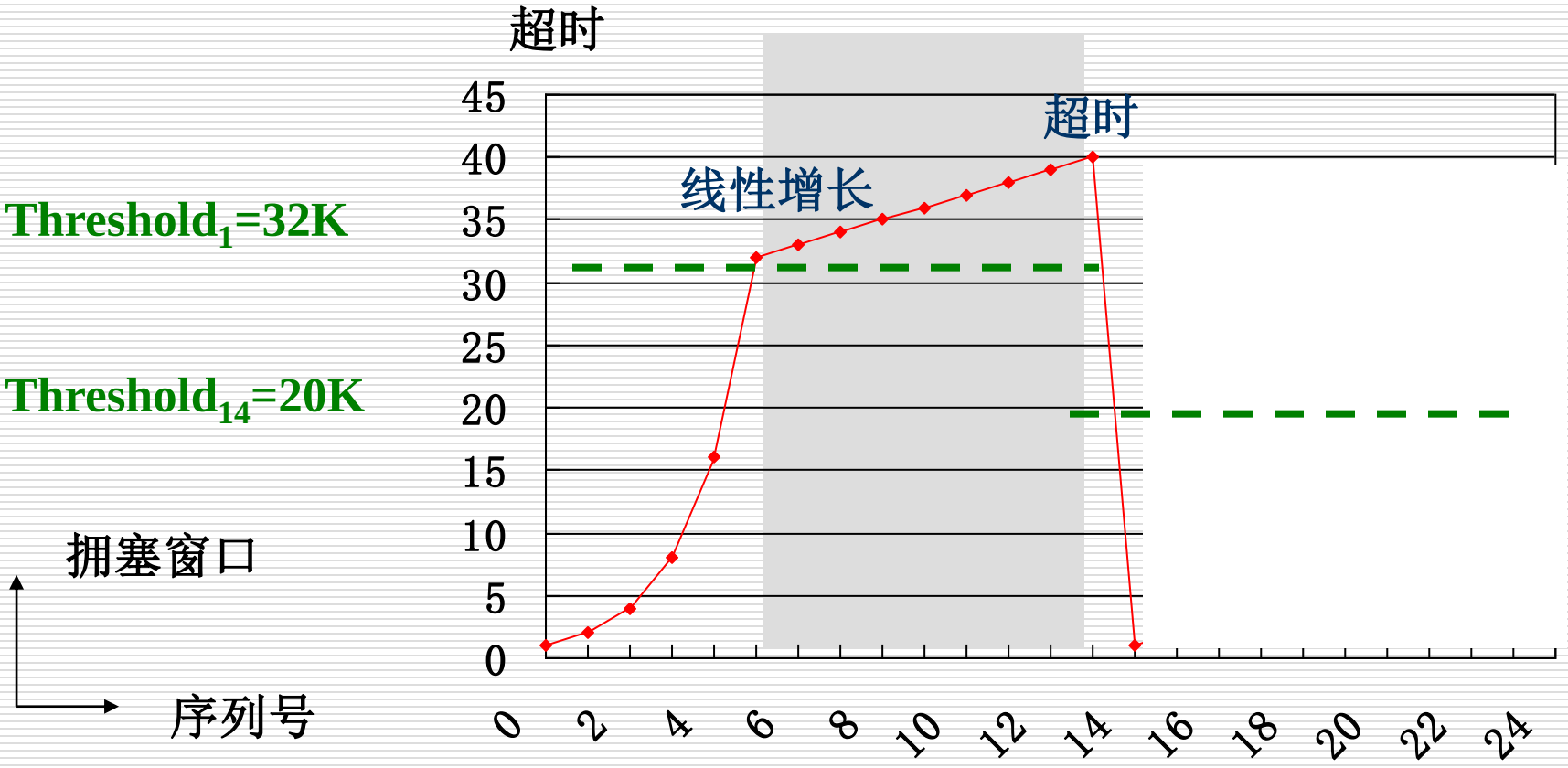
MaxSegL=1024



$Threshold_1$=32K

关键参数（临界值，接收窗口，拥塞窗口）

拥塞控制算法- cwnd线性增长



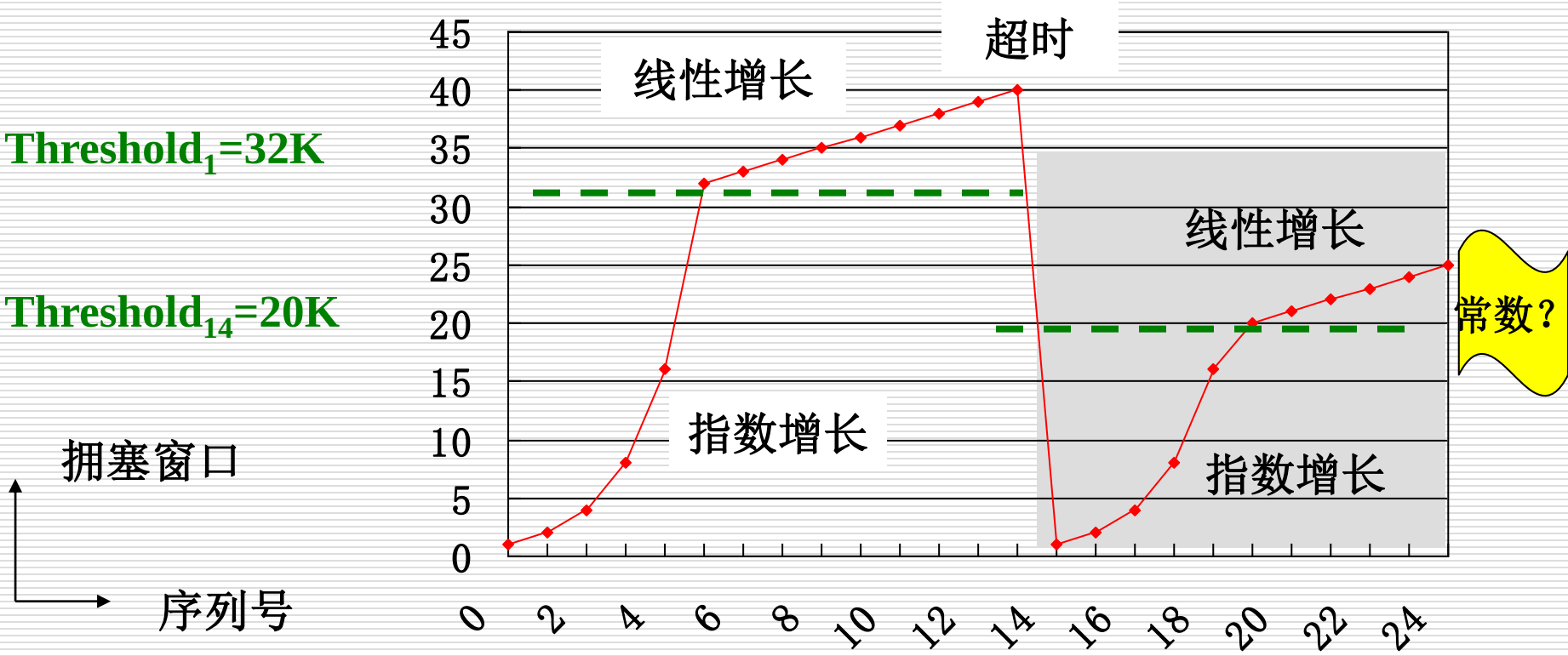
关键参数（临界值，接收窗口，拥塞窗口）

拥塞控制算法-重新慢速启动

$Threshold_0$=64K

超时

MaxSegL=1024



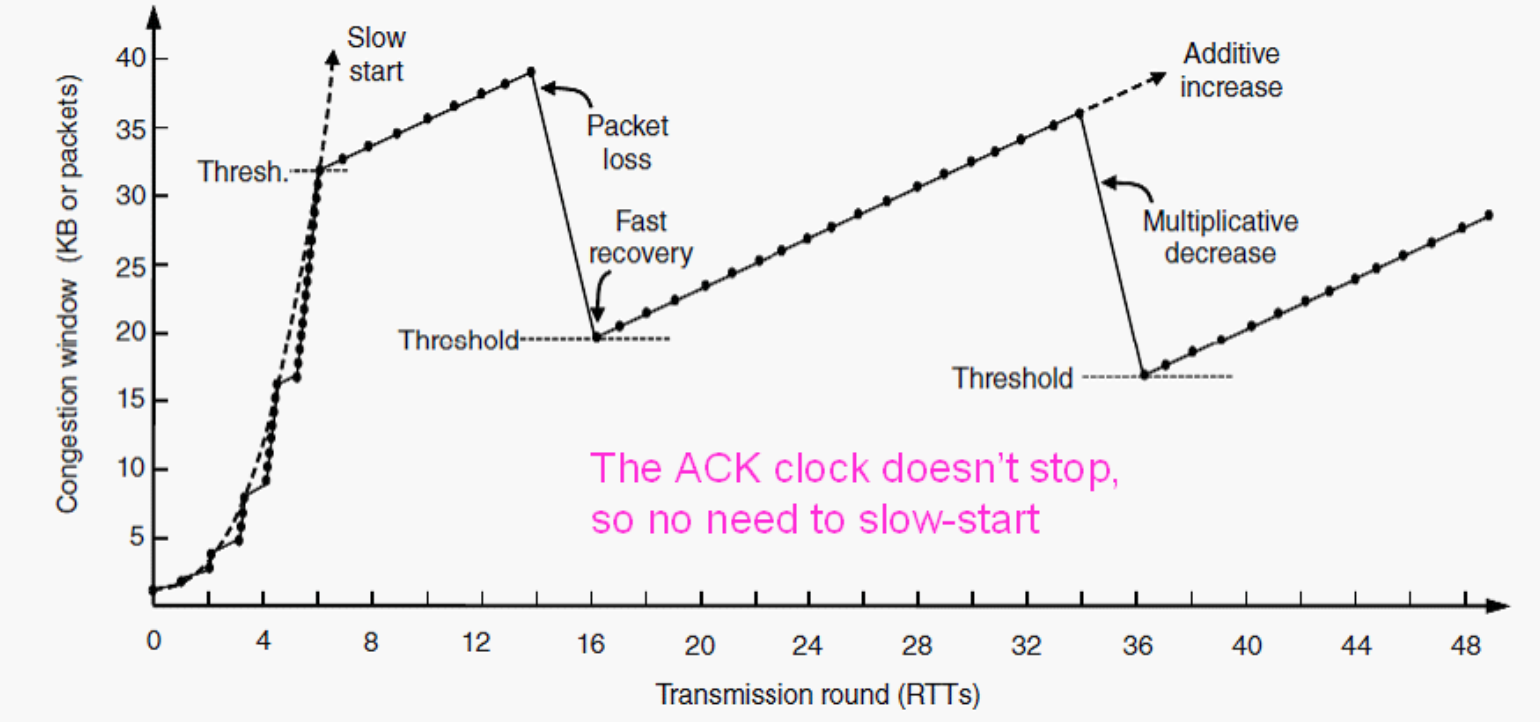
$Threshold_1$=32K

$Threshold_{14}$=20K

关键参数（临界值，接收窗口，拥塞窗口）

注意

快速恢复


Slow start
Thresh.
Packet loss
Fast recovery
Threshold
Additive increase
Multiplicative decrease
Threshold
The ACK clock doesn't stop, so no need to slow-start
Congestion window (KB or packets)
Transmission round (RTTs)
0
4
8
12
16
20
24
28
32
36
40
44
48
5
10
15
20
25
30
35
40

拥塞控制算法

- 定义初始拥塞窗口阈值和窗口大小

  $Threshold_0$和$cwnd_0$

- 初始超时
  - 拥塞窗口阈值减半：

  Threshold1 = CWND / 2

- **cwnd二进制指数增长至确认超时**
- **cwnd线性增长至确认超时**
  - 拥塞窗口值减半：**$Threshold_n = CWND_n / 2$**
  - 定义窗口大小：**$cwnd = cwnd_0$**
- 重新开始慢速启动过程

注意P446

- 如果收到一个ICMP抑制分组（ICMP source quench）并被送给TCP传输实体，则这个事件被当作超时对待

课堂练习

- 如果:TCP慢启动中，初始阈值是 8，当拥塞窗口升到12的时候，发生了超时，开始新的慢启动，拥塞窗口从头起CWND= 1。问：第14次传输时的拥塞窗口是多大？_____

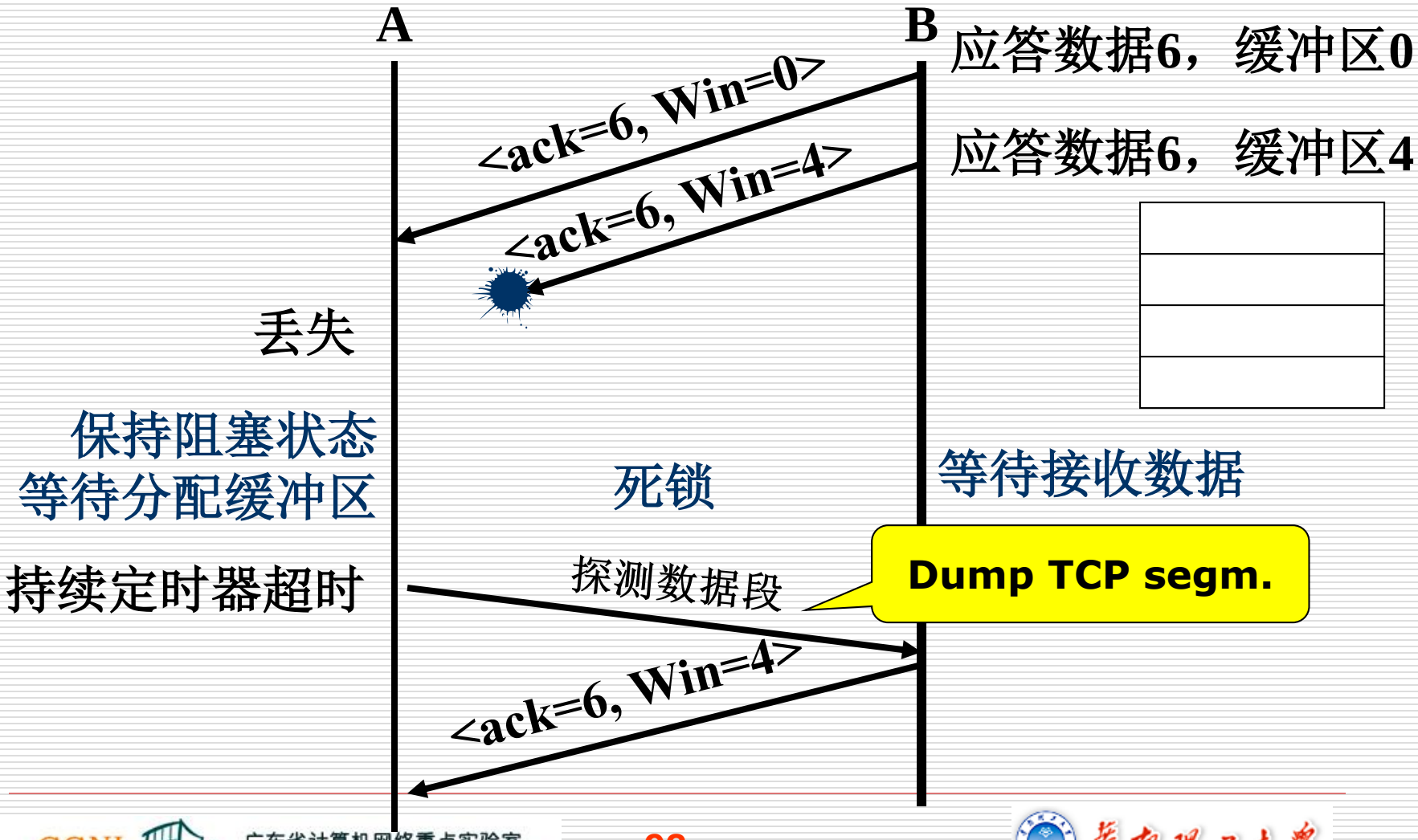
Th=6

TN	1	2	3	4	5	6	7	8	9	10	11	12	13	14
CWND	1	2	4	8	9	10	11	12	1	2	4	6	7	8

TCP 定时器管理 P438

- 最重要的定时器是重传定时器(retransmission timer,Positive ackn. with retransmit）
  - 超时间隔设为多长合适呢？
- 持续定时器（persistence timer），用来避免如下的死锁（ deadlock ）发生
  1. 接收方发送了一个窗口数为零的确认（窗口更新），告诉发送方等待
  2. 稍后，接收方空出了缓冲，发送更新窗口的数据段，但是，很不幸，该分组丢失啦！
  3. 现在，收发双方都在等待对方发送数据段过来，但永远等不到！死锁产生

怎样防止死锁？


A
B
应答数据6，缓冲区0
<ack=6, Win=0>
应答数据6，缓冲区4
<ack=6, Win=4>
丢失
保持阻塞状态
等待分配缓冲区
死锁
等待接收数据
持续定时器超时
探测数据段
Dump TCP segm.
<ack=6, Win=4>

TCP 定时器管理（P439）

- 保活定时器（**keep-alive timer**）用来检查连接是否存活，当一个连接空闲的时间超过保活定时器的时间，该连接将被杀掉。
- 最后一个定时器是在关闭时刻处于**TIMED WAIT** 状态中使用的定时器，它运行两倍的最大分组生存时间，以确保连接关闭之后，该连接上的所有分组都完全消失

比较 TCP 和 UDP

性能	TCP	UDP
可靠性	✓	×
传输延迟	不确定	网络延迟
拥塞控制	✓	×

比较 TCP 和 UDP（续）

- **TCP**
  - 可靠传输方式
  - 可让应用程序简单化，程序员可以不必进行错误检查、修正等工作
- **UDP**
  - 为了降低对计算机资源的需求，如DNS
  - 应用程序本身已提供数据完整性的检查机制，勿须依赖传输层的协议来保证
  - 应用程序传输的并非关键性的数据，如路由器周期性的路由信息交换
  - 一对多方式，必须使用UDP（TCP限于一对一的传送），如视频传播

总结

- UDP (数据段segment)
- TCP (数据段segment)

比较

- 提高可靠传输的措施 (传输策略)
  - 肯定确认重传
  - 窗口技术 (滑窗技术)
  - nagle 算法 和 clark方案
  - 拥塞控制 (慢启动)

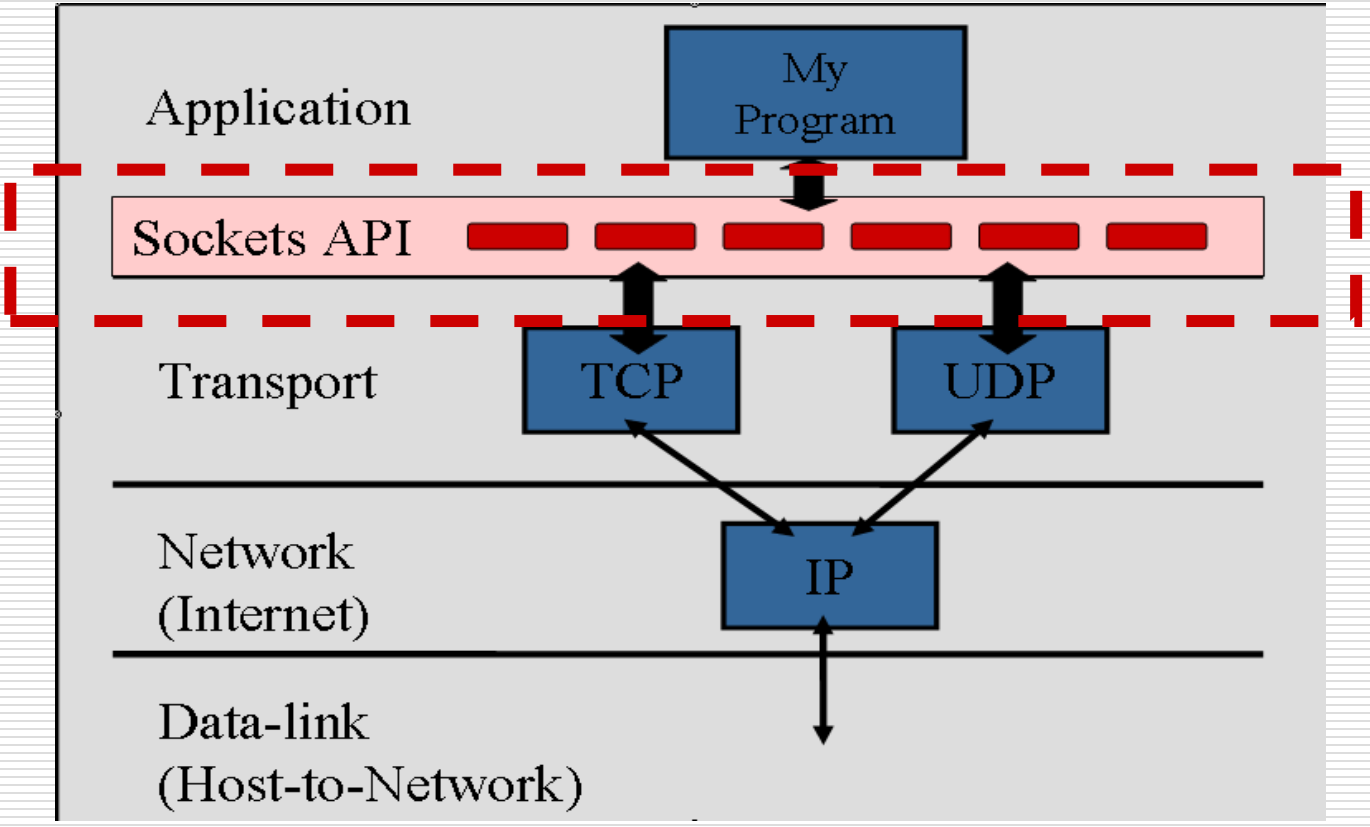
补充：Socket编程简介

- 套接字简介
- 客户/服务器模式的主要接口
- 实验
- 参考资料（推荐）
  - 《UNIX网络编程》richard stevens
  - 《TCP/IP网络互联技术 卷III （winsock版 ）》
  - 《WINDOWS网络编程》微软

套接字简介

- 套接字接口（socket interface）
  - **由加州大学伯克利分校UNIX小组开发，目前最为流行。**
  - **定义了网络上的各种操作（如生成套接字，发送/接收消息等）**
- 常用的套接字接口
  - **Linux/Unix：Berkeley Socket是最突出的一套接口。**
  - **Windows：Win Socket，也称winsock，与BS很类似的接口**

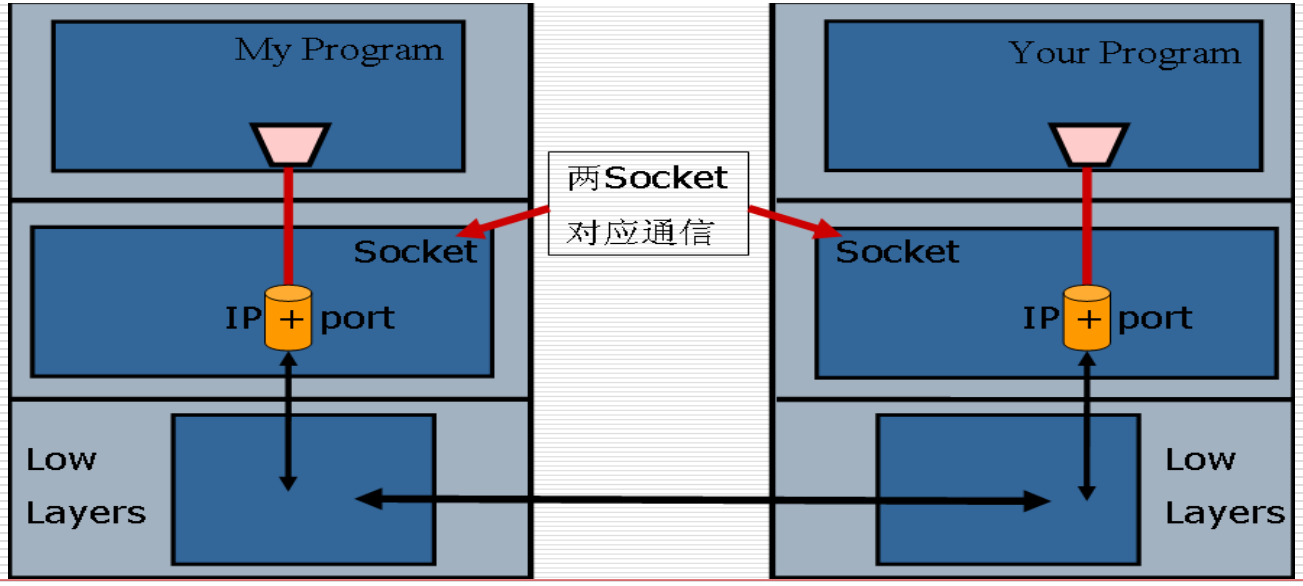
套接字介绍


Application
My Program
Sockets API
Transport
TCP
UDP
Network (Internet)
IP
Data-link (Host-to-Network)

套接字简介（续）

- 对于套接字的简单理解：

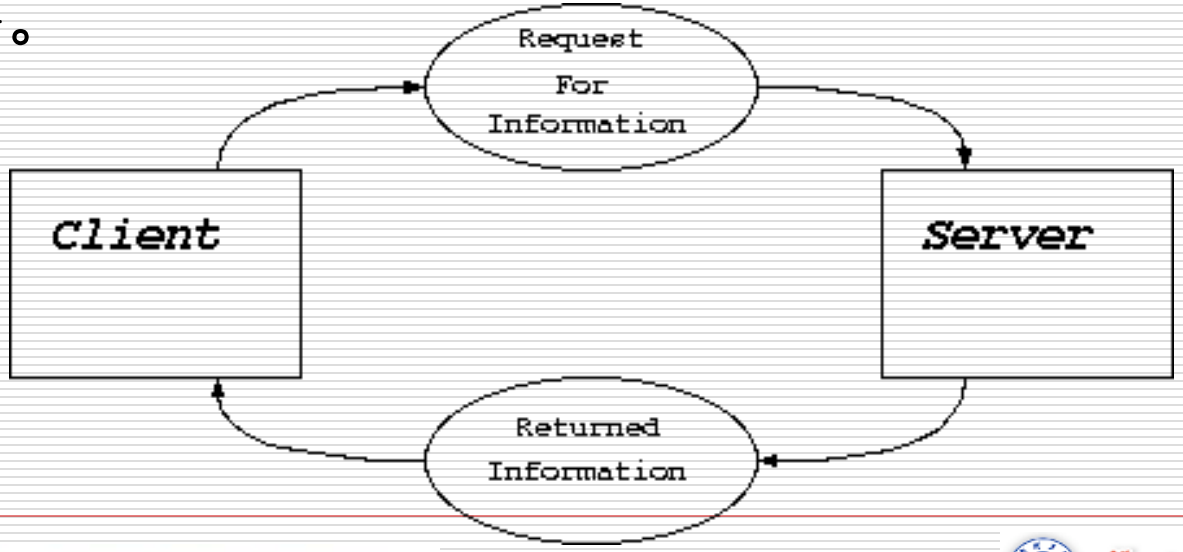
可认为一个IP与一端口（port）联合在一起形成一个套接字，它是网络上的一个传输接口。在网络的另外一端可有一个对应的套接字与通信。



客户/服务器模式

- 客户/服务器模式

TCP/IP网络应用中，最常用的通信模式是客户/服务器模式(Client/Server model)，即客户向服务器发出服务请求，服务器接收到请求后，提供相应的服务。



客户/服务器模式（续）

客户端与服务器的连接方式主要有两种：

- 流式套接口（TCP）

  流式套接口是可靠的双向通讯的数据流。传送的包会按发送时的顺序到达。

  int s=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);

- 数据报套接口（UDP）

  使用这种方式，传送的包不一定会按发送时的顺序到达。

  int s=socket(PF_INET,SOCK_DGRAM,IPPROTO_UDP);

客户/服务器模式-服务器端

- 1.服务器先要端打开一个通信通道，并告知本地主机它需要在某个端口上（如FTP为21）接收客户请求；
- 2.等待客户请求到达该端口；
- 3.接收到服务请求，处理该请求并应答。直至交互完成。
- 4.返回第二步，等待另一客户请求。
- 5.关闭服务器

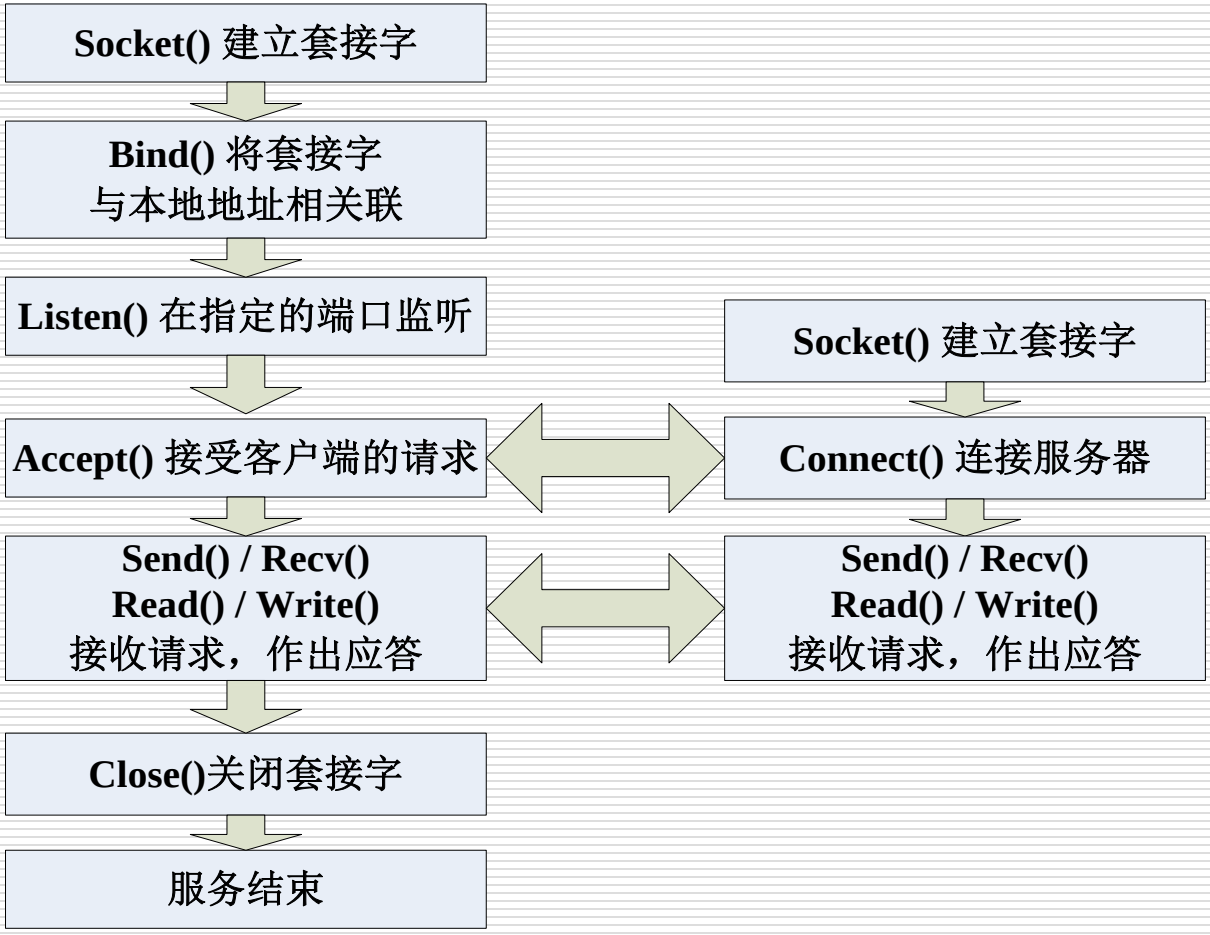
客户/服务器模式-客户端

1. 打开一个通信通道，连接到服务器所在主机的特定端口

   （此时，服务器端已经在这个Socket等待请求）
2. 向服务器发服务请求报文，等待并接收应答；

   继续提出请求并等待应答.....
3. 请求结束后关闭通信通道并终止。

流式套接口的工作流程


Socket() 建立套接字
Bind() 将套接字
与本地地址相关联
Listen() 在指定的端口监听
Accept() 接受客户端的请求
Send() / Recv()
Read() / Write()
接收请求，作出应答
Close()关闭套接字
服务结束
Socket() 建立套接字
Connect() 连接服务器
Send() / Recv()
Read() / Write()
接收请求，作出应答

客户/服务器模式（续）

从上面所描述过程可知：

1. 客户与服务器进程的作用是非对称的，它们各自完成的功能不同，因此编码也不同。
2. 服务进程一般是先于客户请求而启动的，启动后即在相应的Socket监听来自客户端的请求。只要系统运行，该服务进程一直存在，直到正常或强迫终止。

客户/服务器模式需用到的接口

- 服务器方面初始时需要执行的操作：
  - **int socket ()** 建立一个Socket
  - **int bind()** 与某个端口绑定
  - **int listen()** 开始监听端口
  - **int accept()** 等待/接受客户端的连接请
- 客户端需要执行的操作：
  - **int socket ()** 建立一个Socket
  - **int connect()** 连接到服务器

课程作业任务（实验5）

- 实现一个客户端与服务器的互通小程序
  - 服务器端和客户端创建套结字
  - 能够互相传文字（文件、键盘输入或就是固定文字传送）
  - 建议附加功能（可获得加分）：
    - 有界面。
    - 在获取文件之前能够先得到文件列表。
    - 支持IPv6
- 须基于底层的Socket（具体见下页说明），不使用高层封装的Socket（如Java类库，MFC等）

课程作业要求

- 平台要求：Linux和windows均可
- 语言要求：使用C或C++语言
- 环境要求：不限，Visual C++、TuboC、GCC
- 使用库要求：为了让同学们更好地理解Socket的运作
  - Linux平台下只能使用底层库的socket (socket.h)
  - Windows平台下只能使用Winsock (winsock.h)
  - 请勿使用其它高层封装的Socket库（如Java类库，MFC等）

课程作业提交

- 提交时间：2016年12月26日
- 个人独立完成（如想作较大的项目，可自由组合，但不可超过2个人，且需要预先发email给我确认）
- 提交内容：实验报告（包括程序流程，主要的接口调用关系），源代码，可执行程序，以上内容分成三个文件夹存放（分别是Doc、Src、bin），再统一压缩打包提交到学习中心，压缩包名字：学号_姓名。
- 请大家有问题多互相讨论、群内讨论

thanks !

Important terms

- Transport layer：传输层
- Transport entity：传输实体
- Socket：套接字
- Port （number）：端口（号）
- User datagram protocol（UDP）：用户数据报协议
  - UDP segment

Important terms(update)

- **Transmission control protocol（TCP）：传输控制协议**
  - **TCP segment**
  - **Three handshake**
  - **Connection release**
- **Slow start**
- **Timer**