第3章 运输层

華中科技大學
网络空间安全学院
School of Cyber Science and Engineering, HUST

◆学习目标

- ◆了解运输层所提供的服务及原理
  - ◆多路复用、多路分解
  - ◆可靠数据传输
  - ◆流量控制
  - ◆拥塞控制
- ◆学习运输层协议
  - ◆UDP：无连接的运输服务
  - ◆TCP：面向连接的运输服务
  - ◆TCP：拥塞控制

目　　录

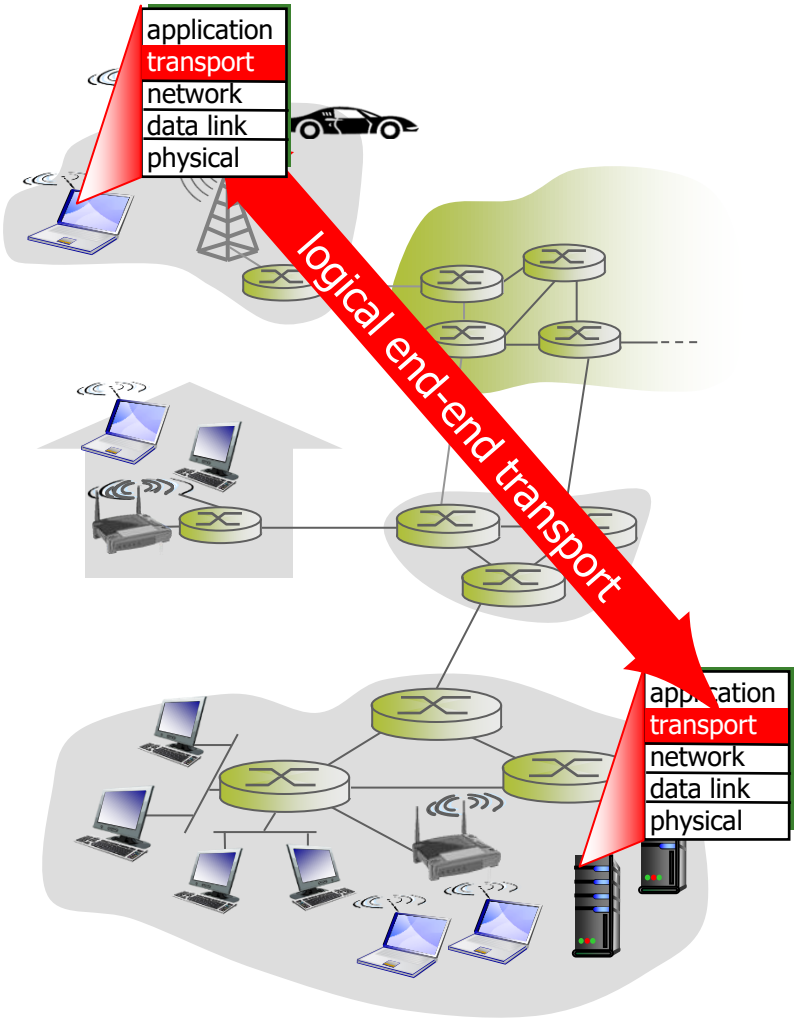
3.1 概述和运输层服务

运输层的功能

- 为不同主机上运行的应用进程之间提供***逻辑通信***(*logical communication*)

运输层协议的工作内容

- 发送方：把应用数据划分成 **报文段**(segments),交给网络层
- 接收方：把**报文段**重组成应用数据，交付给应用层



3.1 概述和运输层服务

运输层和网络层的区别

- ***网络层***: 不同**主机**之间的逻辑通信
- ***运输层***: 应用**进程**之间的逻辑通信

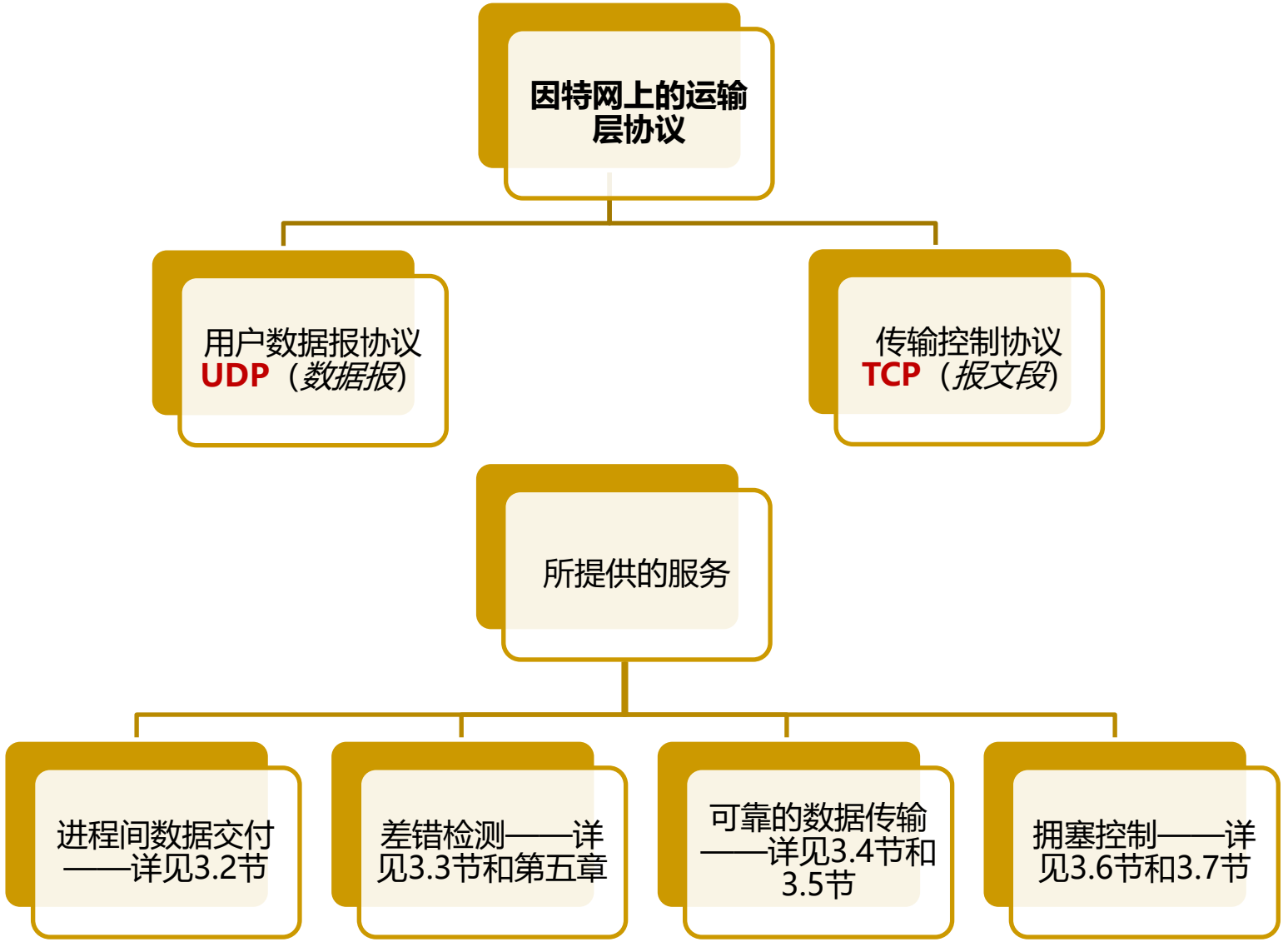
类似于快递:

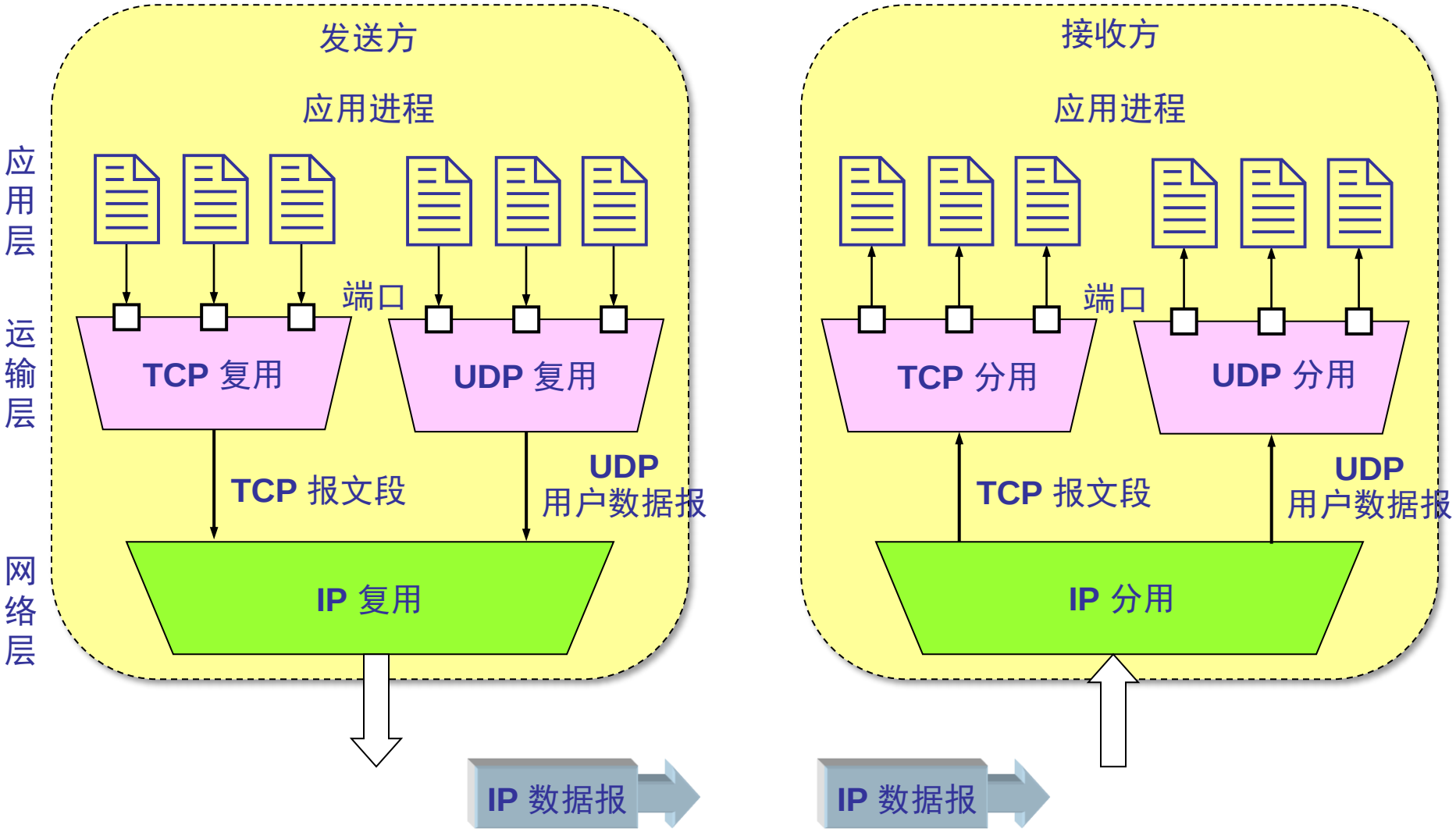
- 一个寝室的学生要给异地另外一个寝室的学生发快递
- 进程 = 学生们
- 进程间报文 = 包裹
- 主机 = 寝室
- 运输协议 = 快递员张三 和 快递员李四
- 网络层协议 = 物流公司提供的服务

◆上例中的几种特殊场景

- ◆张三和李四生病了，无法工作，换成张五和李六
  - ◆ 不同的运输层协议可能提供不一样的服务
- ◆物流公司不承诺信件送抵的最长时间
  - ◆ 运输层协议能够提供的服务受到底层网络协议的服务模型的限制
- ◆物流公司不承诺包裹一定安全可靠的送达，可能在路上丢失，发送者在较长时间内没有受到对方的回执时，再次包装包裹，寄出
  - ◆ 在网络层不提供某些服务的情况下，运输层自己提供

3.1 概述和运输层服务


因特网上的运输层协议
用户数据报协议
UDP（数据报）
传输控制协议
TCP（报文段）
所提供的服务
进程间数据交付——详见3.2节
差错检测——详见3.3节和第五章
可靠的数据传输——详见3.4节和3.5节
拥塞控制——详见3.6节和3.7节

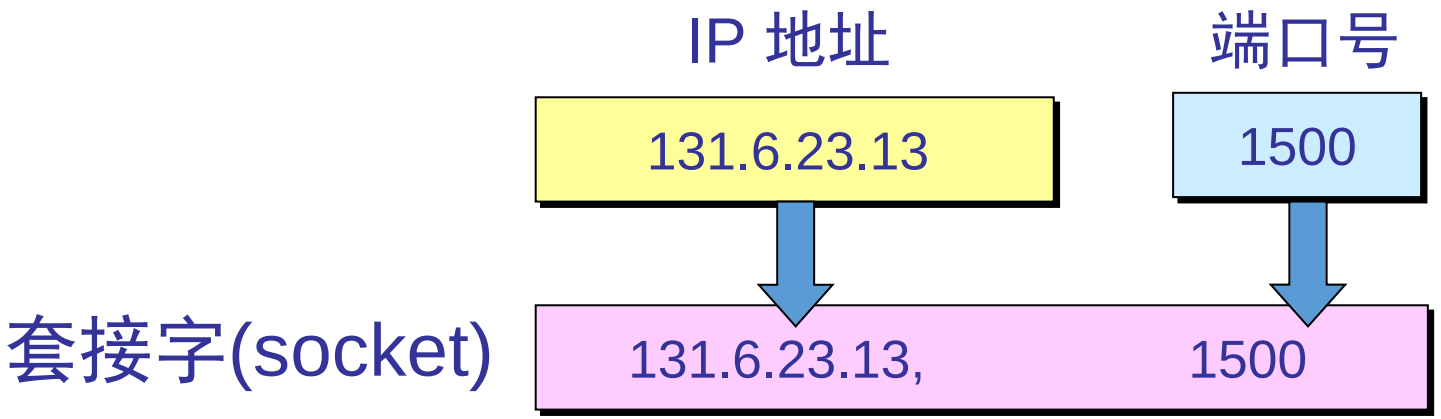

发送方
应用进程
应用层
运输层
网络层
端口
TCP 复用
UDP 复用
TCP 报文段
UDP
用户数据报
IP 复用
IP 数据报
接收方
应用进程
端口
TCP 分用
UDP 分用
TCP 报文段
UDP
用户数据报
IP 分用
IP 数据报

◆端口

- ◆端口的作用就是让应用层的各种应用进程都能将其数据通过端口向下交付给运输层，以及让运输层知道应当将其报文段中的数据向上通过端口交付给应用层相应的进程（或者线程）
- ◆从这个意义上讲，端口是用来标志应用层的进程（或者线程）
- ◆端口用一个 16 bit 端口号进行标志

套接字

- TCP 使用“连接”(而不仅仅是“端口”)作为最基本的抽象，同时将 TCP 连接的端点称为套接字(socket) 。
- 套接字和端口、IP 地址的关系是：



报文段（数据报）的投送

- 主机收到IP包
  - 每个数据包都有源IP地址和目的IP地址
  - 每个数据包都携带一个传输层的数据报文段
  - 每个数据报文段都有源、目的端口号
- 主机根据“IP地址＋端口号”将报文段定向到相应的套接字

TCP/UDP 报文段格式

•无连接的情形

- *recall*：创建的套接字具有主机本地端口#：

```
DatagramSocket mySocket1
 = new DatagramSocket(12534);
```

- *recall*：创建数据报发送到UDP套接字时，必须指定：
  - 目的地IP地址
  - 目的端口 #
  - （单组套接字）

- 当主机接收到UDP段：
  - 检查报文段目的地端口#
  - 将UDP报文段发送给端口号#的套接字

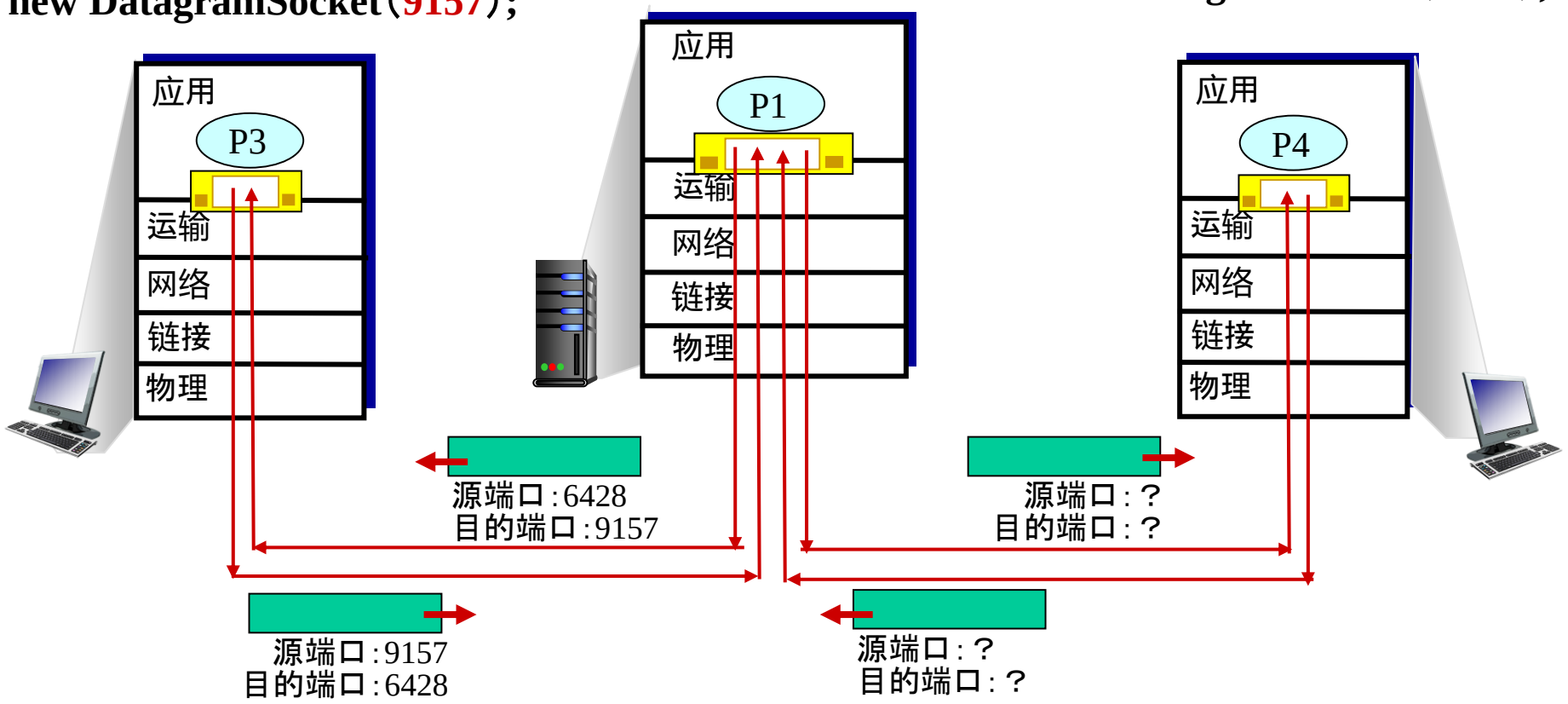
→ IP数据报 *相同的DEST.port #*，但不同的源IP地址 *和/或* 源端口号将被发送到**目的地址相同的套接字**

对于UDP来说，发给同一个进程的数据，无论是从哪里来，都只用一个套接字来接收

DatagramSocket serverSocket =
new DatagramSocket(6428);

DatagramSocket的mySocket2 =
new DatagramSocket(9157);

DatagramSocket mySocket1 =
new DatagramSocket(5775);



socket发送报文（C语言）


```
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <arpa/inet.h>

void main()
{
        struct sockaddr_in dest_info;
        char * data = "Hello World!";

      //create a network socket
      int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

      memset((char *)&dest_info, 0 ,sizeof(dest_info));
      dest_info.sin_family = AF_INET;
      dest_info.sin_addr.s_addr = inet_addr("127.0.0.1");
      dest_info.sin_port = htons(9090);

      sendto(sock, data, strlen(data), 0 , (struct sockaddr*)&dest_info, sizeof(dest_info));
      close(sock);

}
```

socket接收报文（C）


```c
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <arpa/inet.h>

void main()
{
      struct sockaddr_in server;
      struct sockaddr_in client;
      int clientlen;
      char buf[1500];

      //create a network socket
      int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);

      memset((char *)&server, 0 ,sizeof(server));
      server.sin_family = AF_INET;
      server.sin_addr.s_addr = htonl(INADDR_ANY);
      server.sin_port = htons(9090);

      if( bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0){
           printf("Error on binding");
           close(sock);
           return;
        }

      while(1){
         bzero(buf,1500);
         recvfrom(sock, buf, 1500-1, 0, (struct sockaddr*)&client, &clientlen);
         printf("received message: %s\n",buf);
      }

      close(sock);

}
```

```
[03/23/22]seed@VM:~$ vi udp_s.c
[03/23/22]seed@VM:~$ ./udp_s
received message: hello

received message: hahaha
```

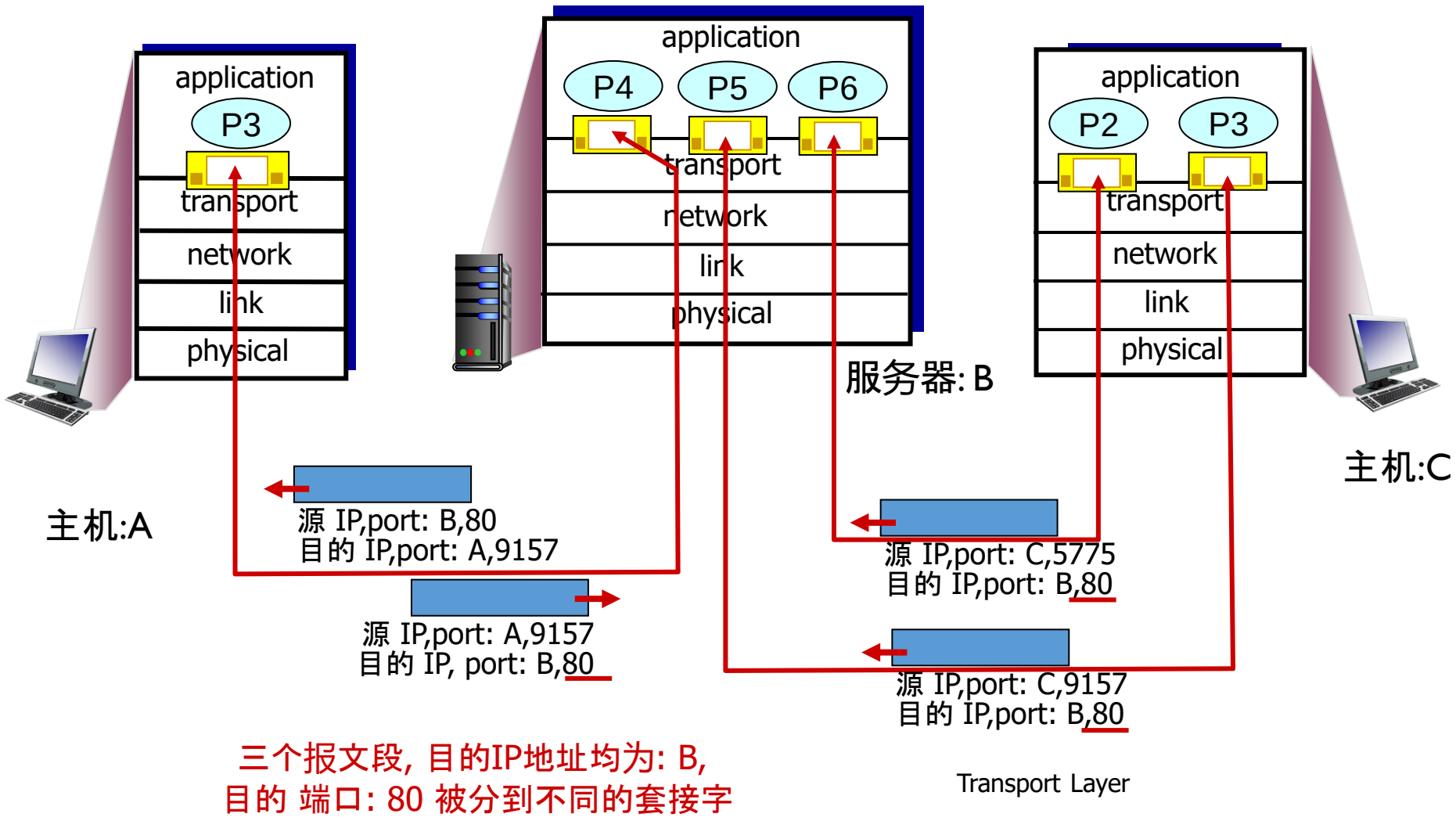
```
[03/23/22]seed@VM:~$ nc -u 127.0.0.1 9090
hello
hahaha
```

面向连接的复用和分用

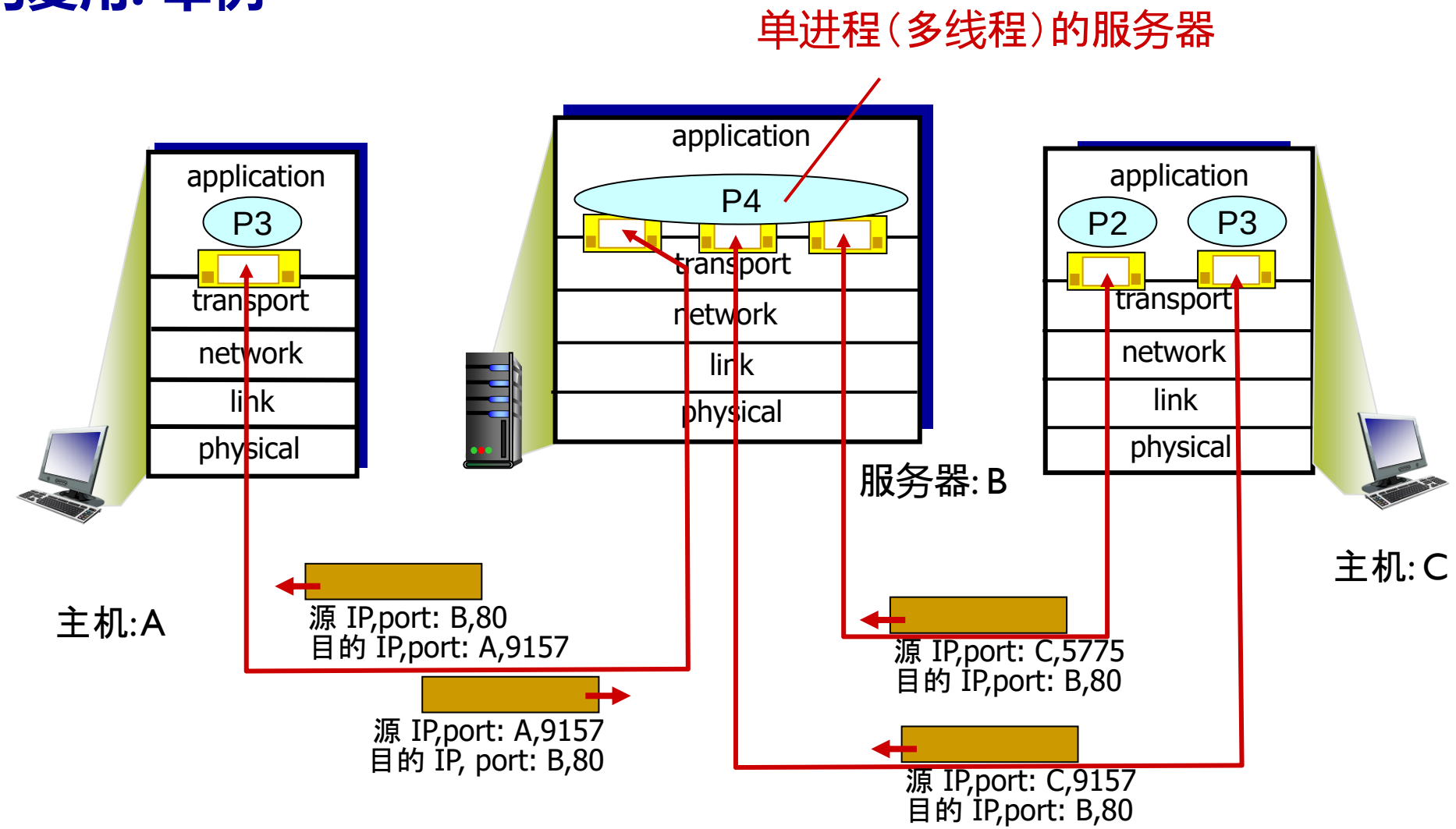
- TCP 套接字由一个四元组来标识（**一个连接一组套接字**）

  （源IP地址，源端口号，目的IP地址，目的端口号）

- 接收方主机根据这四个值将报文段定向到相应的套接字
- 服务器主机同时支持多个并发的TCP套接字:
  - 每一个套接字都由其四元组来标识
- Web服务器为每一个客户连接都产生不同的套接字
  - 非持久HTTP对每一个请求都建立不同的套接字（会影响性能）


application
P3
transport
network
link
physical
主机:A
application
P4
P5
P6
transport
network
link
physical
服务器: B
application
P2
P3
transport
network
link
physical
主机:C
源 IP,port: B,80
目的 IP,port: A,9157
源 IP,port: A,9157
目的 IP, port: B,80
源 IP,port: C,5775
目的 IP,port: B,80
源 IP,port: C,9157
目的 IP,port: B,80
三个报文段, 目的IP地址均为: B,
目的 端口: 80 被分到不同的套接字
Transport Layer

面向连接的复用: 举例


单进程(多线程)的服务器
application
P3
transport
network
link
physical
主机:A
application
P4
transport
network
link
physical
服务器: B
application
P2
P3
transport
network
link
physical
主机: C
源 IP,port: B,80
目的 IP,port: A,9157
源 IP,port: A,9157
目的 IP, port: B,80
源 IP,port: C,5775
目的 IP,port: B,80
源 IP,port: C,9157
目的 IP,port: B,80

TCP socket接收报文（C）


```c
int main(int argc, char *argv[])
{
        int ss,sc;      /*ss为服务器的socket描述符，sc为客户端的socket描述符*/
        struct sockaddr_in server_addr; /*服务器地址结构*/
        struct sockaddr_in client_addr; /*客户端地址结构*/
        int err;                        /*返回值*/
        pid_t pid;                      /*分叉的进行ID*/

        /*建立一个流式套接字*/
        ss = socket(AF_INET, SOCK_STREAM, 0);

        /*设置服务器地址*/
        bzero(&server_addr, sizeof(server_addr));               /*清零*/
        server_addr.sin_family = AF_INET;                       /*协议族*/
        server_addr.sin_addr.s_addr = htonl(INADDR_ANY);        /*本地地址*/
        server_addr.sin_port = htons(PORT);                     /*服务器端口*/

        /*绑定地址结构到套接字描述符*/
        err = bind(ss, (struct sockaddr*)&server_addr, sizeof(server_addr));

        /*设置侦听*/
        err = listen(ss, BACKLOG);

        /*主循环过程*/
        for(;;) {
                socklen_t addrlen = sizeof(struct sockaddr);

                sc = accept(ss, (struct sockaddr*)&client_addr, &addrlen);
                /*接收客户端连接*/
                if(sc < 1){     /*出错*/
                        continue;       /*结束本次循环*/
                }

                /*建立一个新的进程处理到来的连接*/
                pid = fork();           /*分叉进程*/
                if( pid == 0 ){         /*子进程中*/
                        process_conn_server(sc);                /*处理连接*/
                        close(ss);      /*在子进程中关闭服务器的侦听*/
                }else{
                        close(sc);      /*在父进程中关闭客户端的连接*/
                }
        }
```

实际上这就是UDP所提供的功能（RFC 768）

- **UDP处理数据的流程**
  - 发送方
    - 从**应用进程**得到**数据**
    - 附加上为多路复用/多路分解所需的**源和目的端口号**及**差错检测信息**，形成**报文段**（*数据报*）
    - 递交给**网络层**，尽力而为的交付给**接收主机**
  - 接收方
    - 从**网络层**接收**报文段**（*数据报*）
    - 根据目的**端口号**，将数据交付给相应的**应用进程**

UDP通信事先无需握手，是无连接的

UDP的优势

- **无需建立连接**——建立连接会增加时延
- **简单**——发送方和接收方无需维护连接状态
- **段首部开销小**——TCP:20Byte vs UDP:8Byte
- **无拥塞控制**——UDP 可按需要随时发送

部分采用UDP协议的应用

- 远程文件服务器（NFS）
- 流式多媒体
- 因特网电话
- 网络管理（SNMP）
- 选路协议（RIP）
- 域名解析（DNS）

UDP大量应用可能导致的严重后果

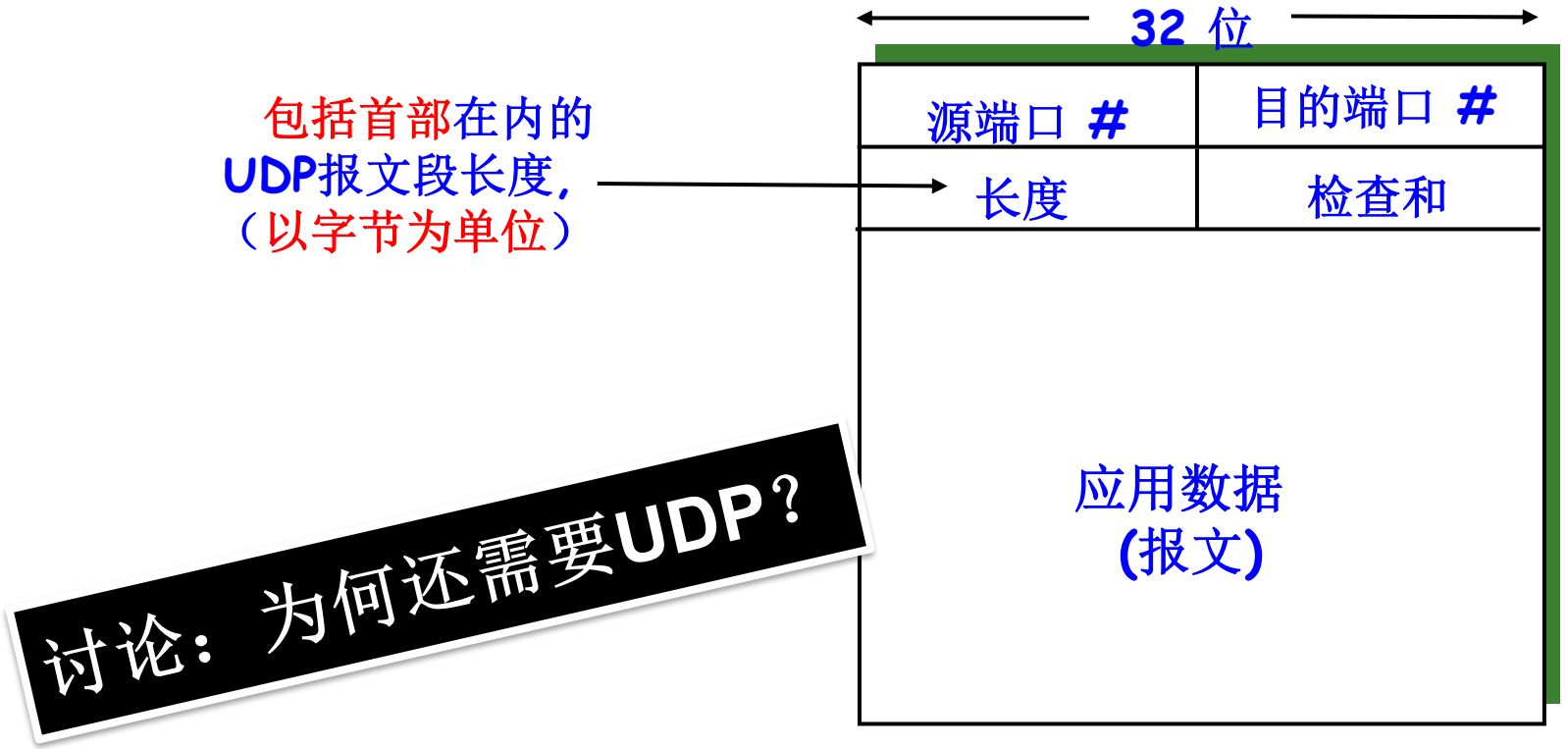
- 路由器中大量的分组溢出
- 显著减小TCP通信的速率，甚至挤垮TCP会话

使用UDP的可靠数据传输

- 在应用层实现数据的可靠传输
- 应用程序特定的错误恢复

增加了应用进程的实现难度

◆UDP报文段（*数据报*）的结构



UDP的检验和

- 目标
  - 检测收到的报文段的“**差错**” （例如，出现突变的比特）
- 发送方
  - 把报文段看作是16**比特字**的序列
  - **检验和**：对报文段的所有16比特字的和进行**反码运算**
  - 发送方将校验和写入UDP**检验和字段**中
- 接收方
  - **计算**接收到的报文段的**和（包括发送方的校验和）**
    - **不为全1--检测除错误**
    - **为全1—没有检测到错误（但仍可能存在错误）**

◆例子: 将两个16比特字相加

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
回卷	(1)	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
和	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
校验和	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

注意： 最高有效位的进位要回卷加到结果当中

是否连首部一起校验？校验字段还没有计算出来呀！

把检验和字段置为0

接收方怎么校验？也用相同的方式计算一遍？但接收方收到的数据是包含校验和阿！

接收方按直接相加的方式计算一遍，结果应该为全1

有没有可能出现奇数字节的UDP数据包？这样的话最后一个字节该怎么计算？

完全有可能，最后一个字节和一个全0的字节计算

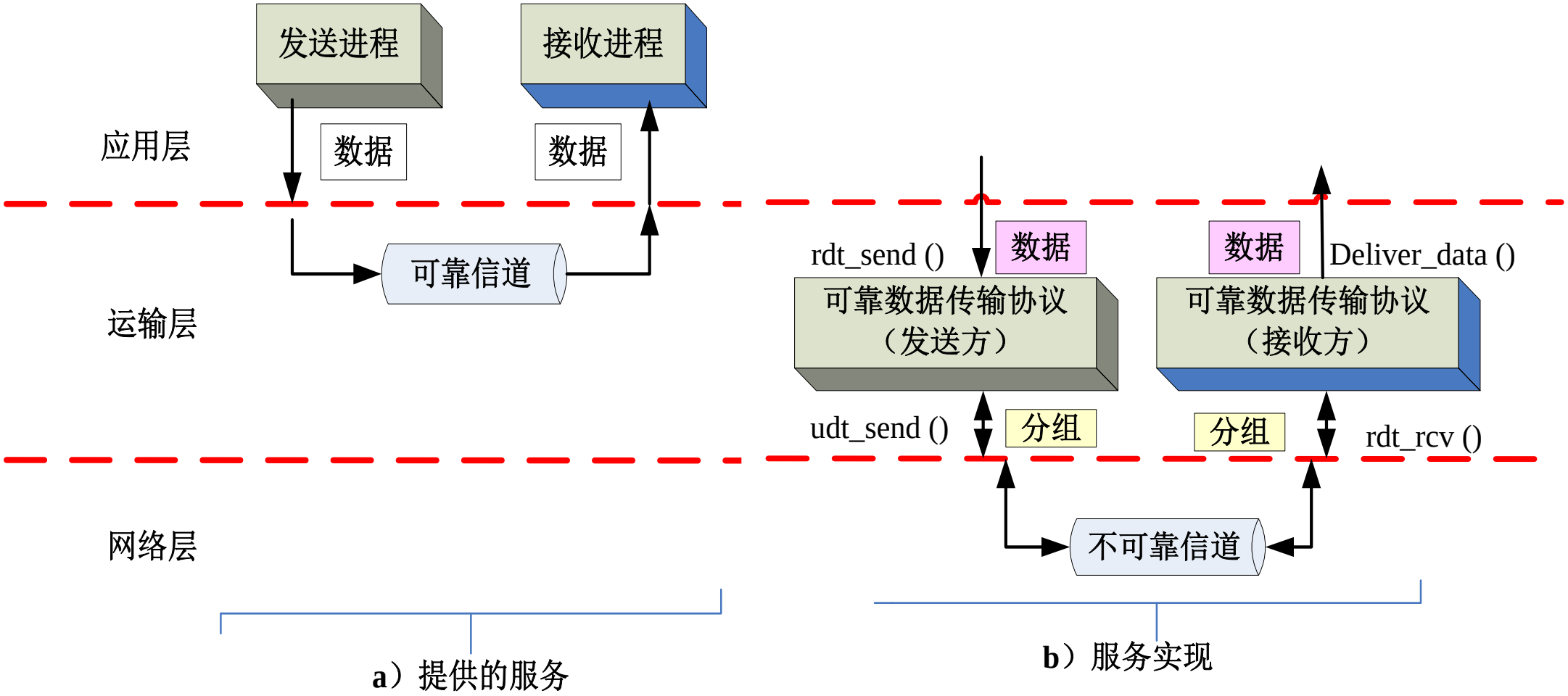
有没有可能出现奇数位（bit）的UDP数据包？这样的话又该怎么计算？

完全不可能，应用层下来的数据一定是按字节的，所以首部是按字节统计长度的

可靠数据传输

- 在应用层、运输层和链路层都很重要
- 网络中最重要的top-10问题之一!

3.4 可靠数据传输的原理

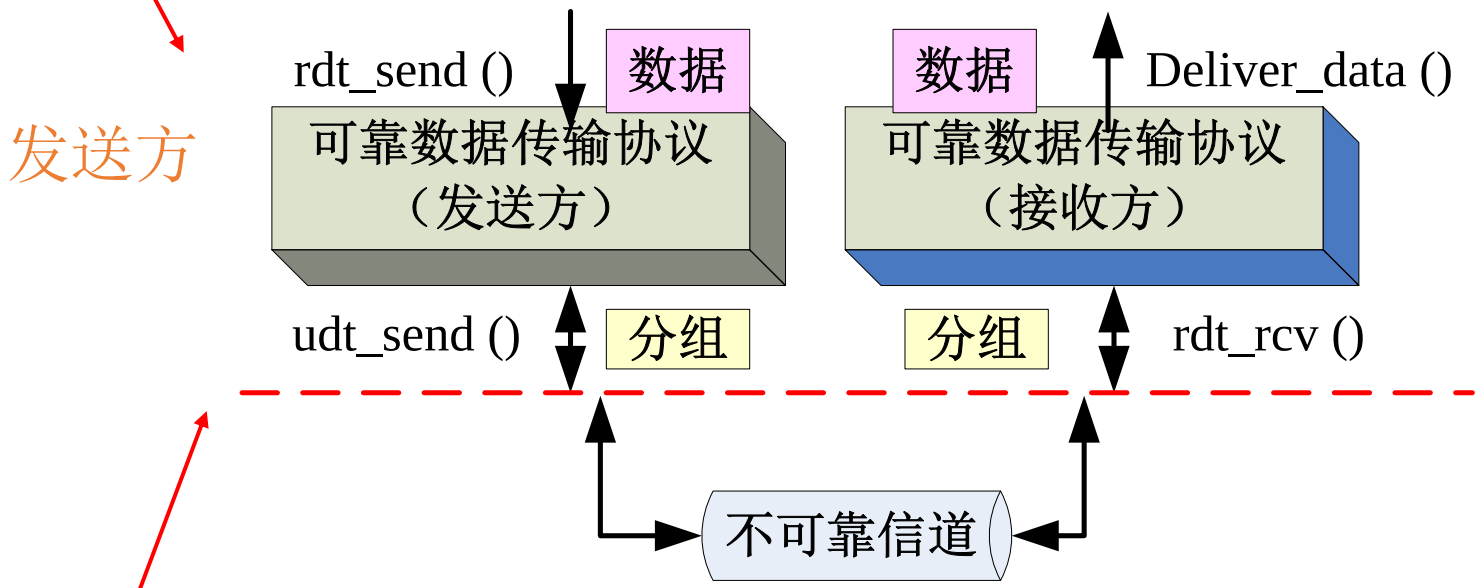


不可靠信道的特性决定了可靠数据传输协议(rdt)的复杂性。

3.4 可靠数据传输的原理

rdt_send()：由上层（如应用层）调用，将数据发送给接收方的上层

deliver_data()：由 **rdt** 调用，将数据交付上层



udt_send()：由 **rdt**调用，将分组通过不可靠通道传给接收方

rdt_rcv()：当分组到达接收方时调用

3.4 可靠数据传输的原理

我们将要:

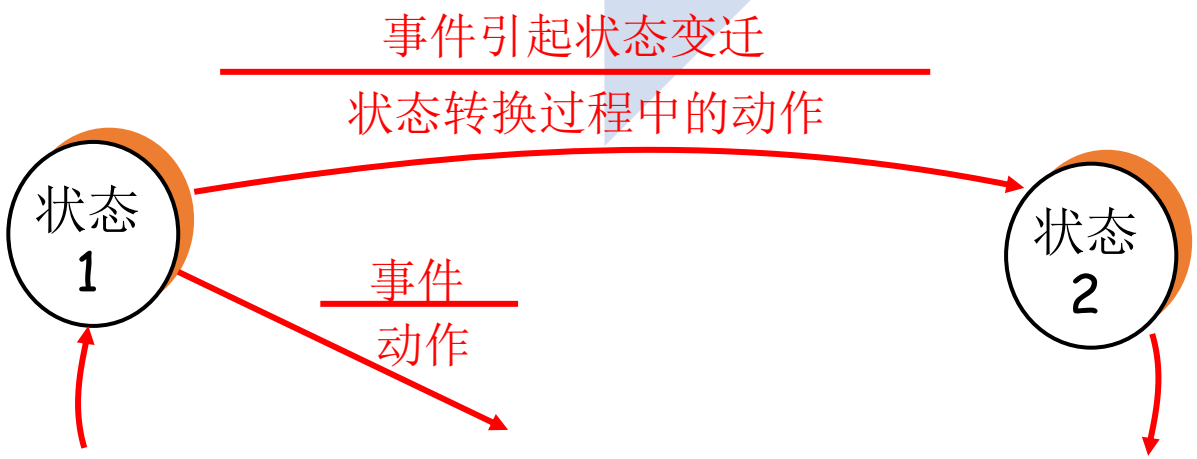
逐步地开发可靠数据传输协议(rdt)的发送方和接收方

只考虑单向数据传输(unidirectional data transfer)的情况

- *但控制信息是双向传输的!*

用有限状态机(FSM) 来描述发送方和接收方

状态: 由事件引起一个状态到另一个状态的变迁。



3.4 可靠数据传输的原理

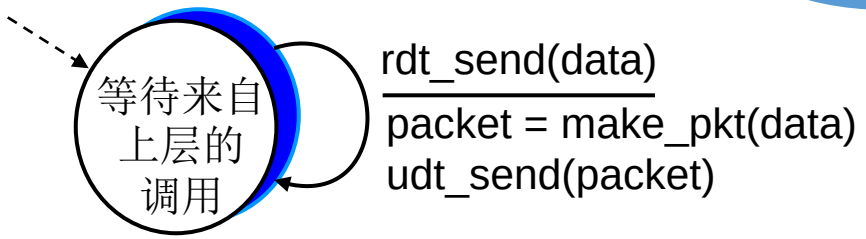
底层信道完全可靠

- 不会产生比特错误
- 不会丢失分组

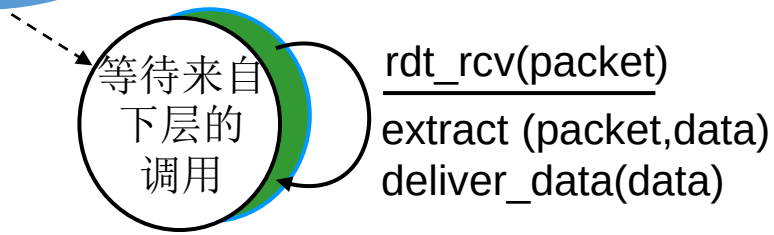
可靠信道上的可靠传输——rdt 1.0

分别为发送方和接收方建立FSM

- 发送方将数据发送给底层信道
- 接收方从底层信道接收数据

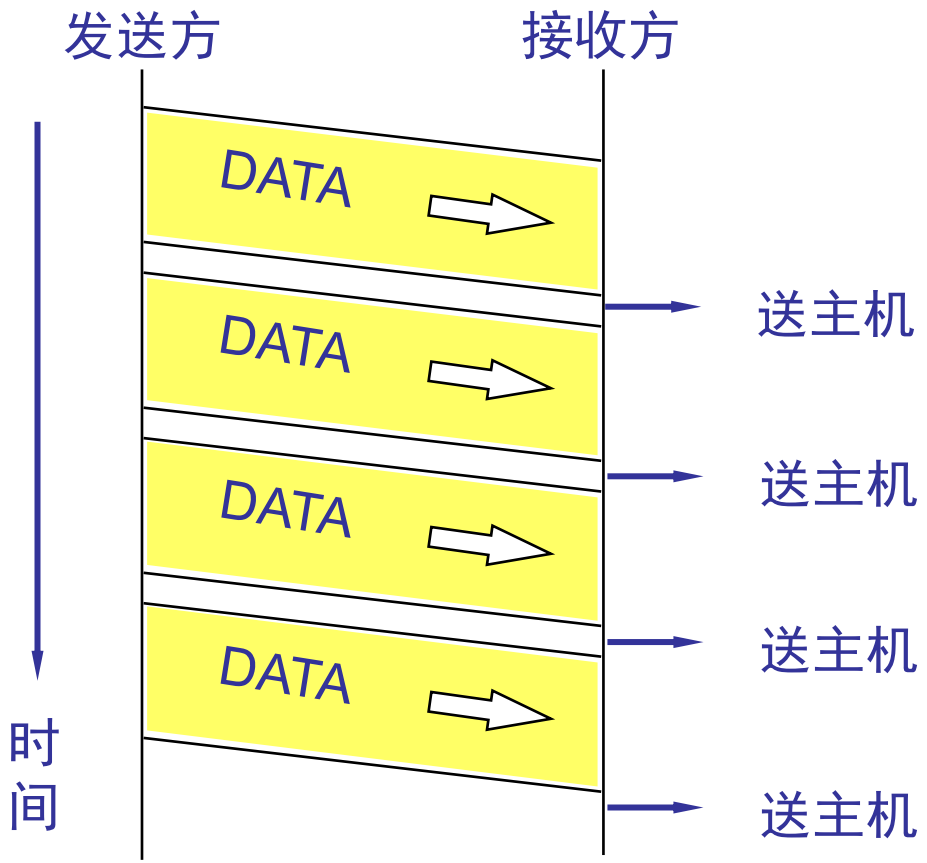


发送方



接收方

- Rdt1.0 时序图



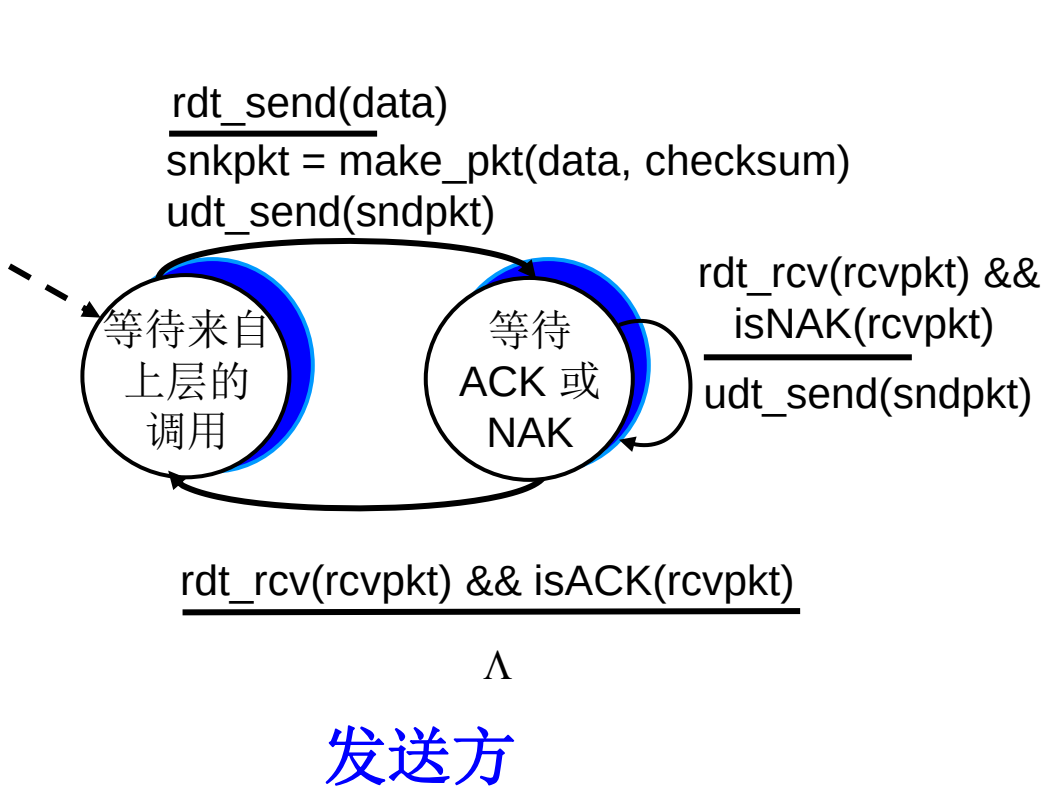
信道可能导致比特出现差错时——rdt 2.x

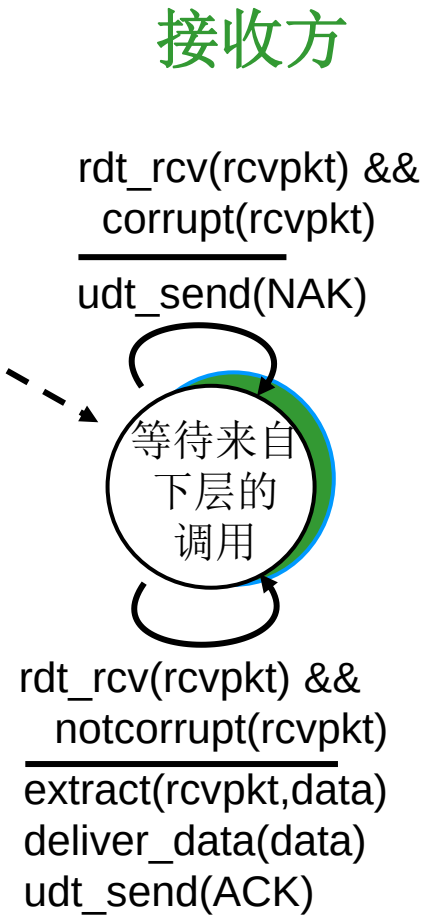
- 第一个版本——rdt 2.0
  - 假设
    - 分组比特可能**受损**
    - 所有传输的分组都将**按序**被接收，不会丢失
  - 处理机制
    - 如何**判断**分组受损——**差错检测**

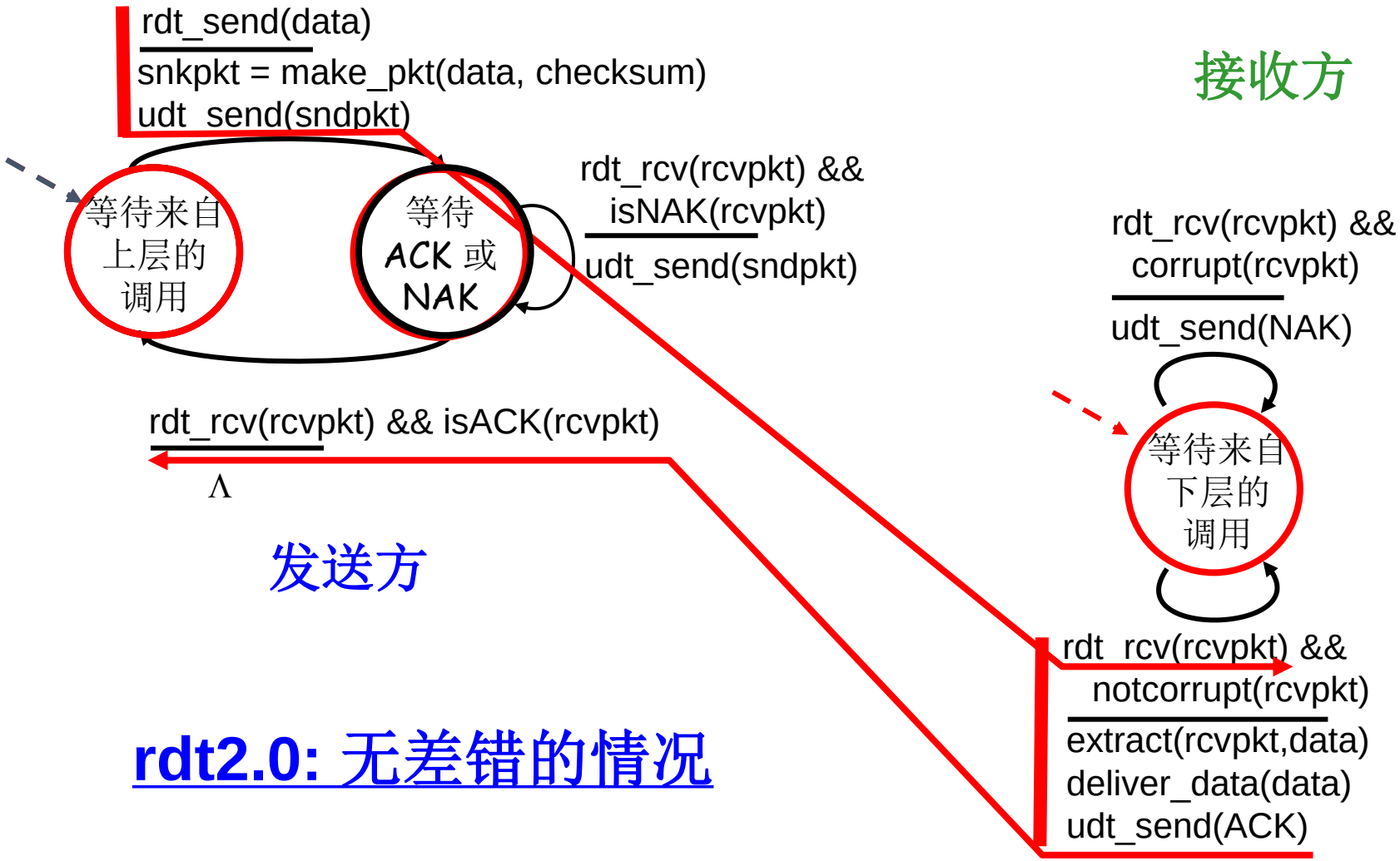
出错了怎么办？--*人类交流时如何处理*

- 如何通知发送方分组是否受损——**接收方反馈**（ACK和NAK）
- 在得知分组受损后，发送方如何处理——**出错重传**

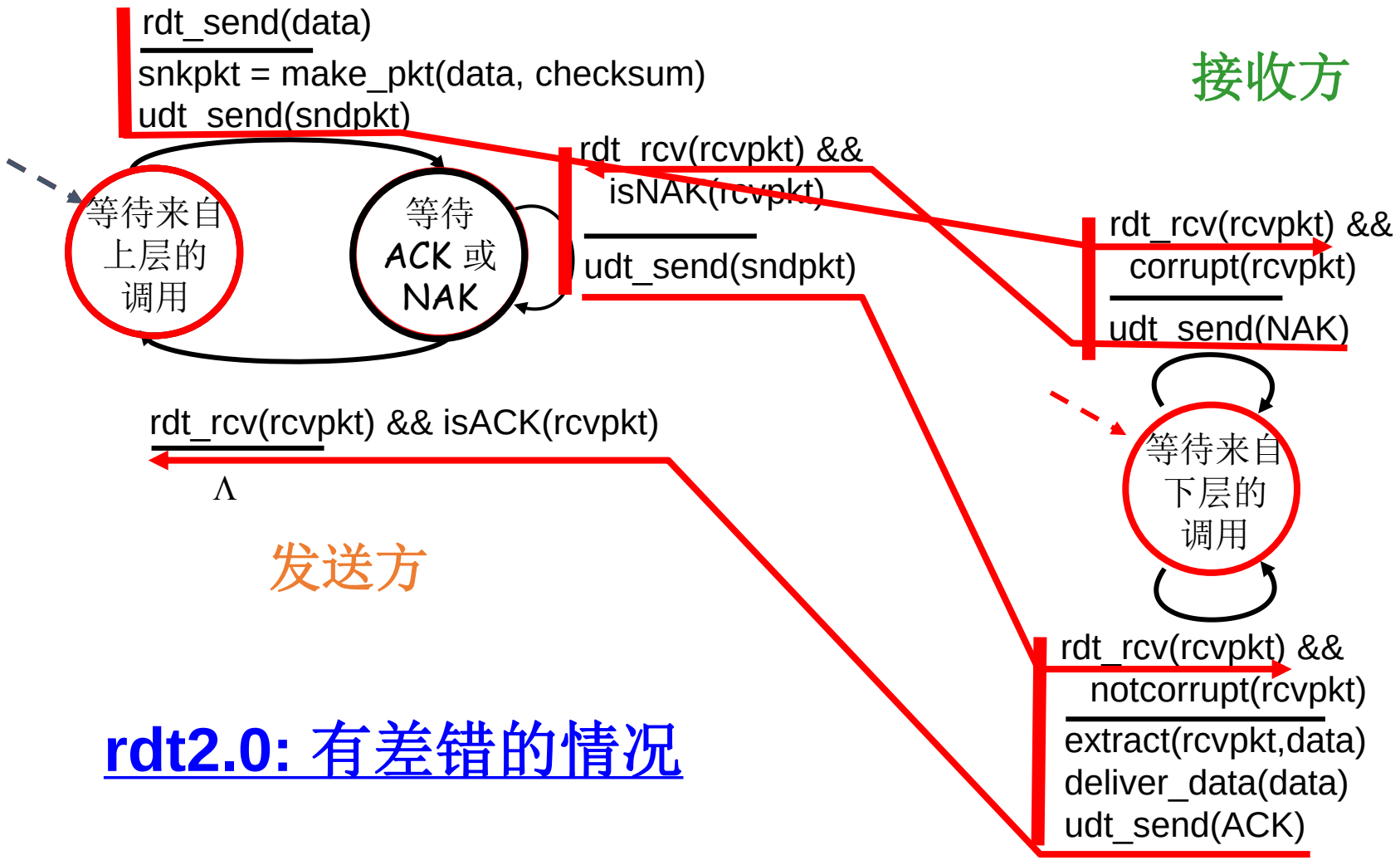
- rdt 2.0的有限状态机FSM







rdt2.0: 无差错的情况



rdt2.0: 有差错的情况

如何实现重传

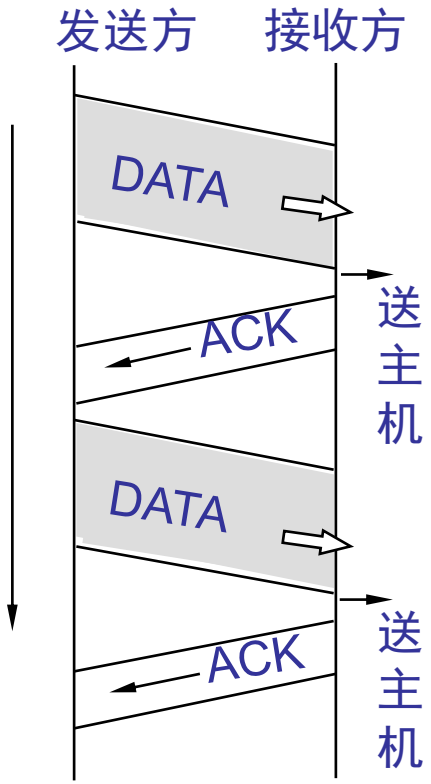
- 使用**缓冲区**缓存已发出但未收到反馈的报文段

接收方和发送方各**一个**报文段大小的缓冲区即可

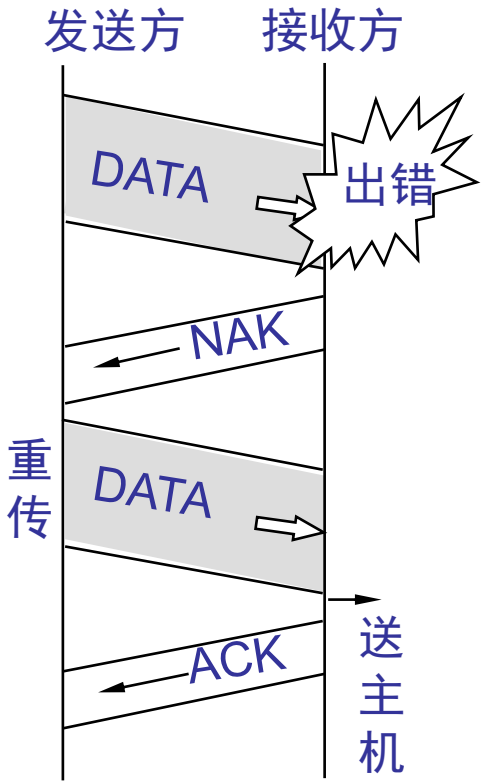
新的问题

- 需要**多大**的缓冲区呢?

Rdt2.0：时序图



(a) 正常情况



(b) 数据出错

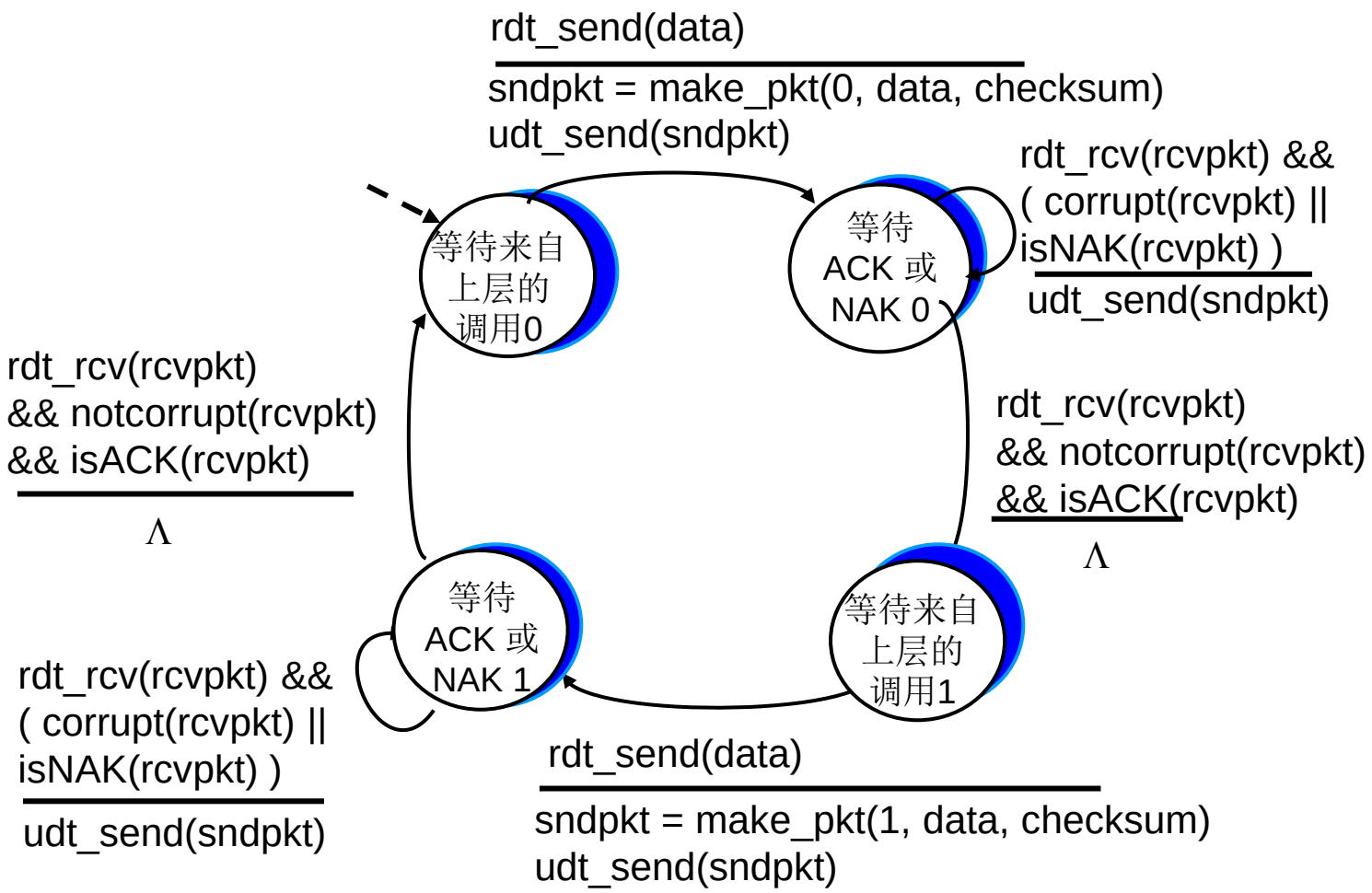
第二个版本——rdt 2.1

- 问题的引入
  - **ACK和NAK分组可能受损**，而rdt 2.0没有考虑该情况—**2.0的一个致命的缺陷**
- 解决问题的几种思路
  - 在人类的对话中，如果听不清楚对方所述，会回问一句“刚才你说什么来着？”但如果这句话仍然没有听清楚呢？怎么办？双方对着问“刚才你说什么来着？”这就可能进入了一个难以解困的死循环
  - 增加足够的检查和比特，使发送方不仅可以检查比特差错，还可以恢复比特差错
  - 收到出错的反馈时，不管三七二十一，直接重发当前数据分组
    - 怎么区分新旧数据?
    - 但这就需要对数据分组进行编号，以示识别

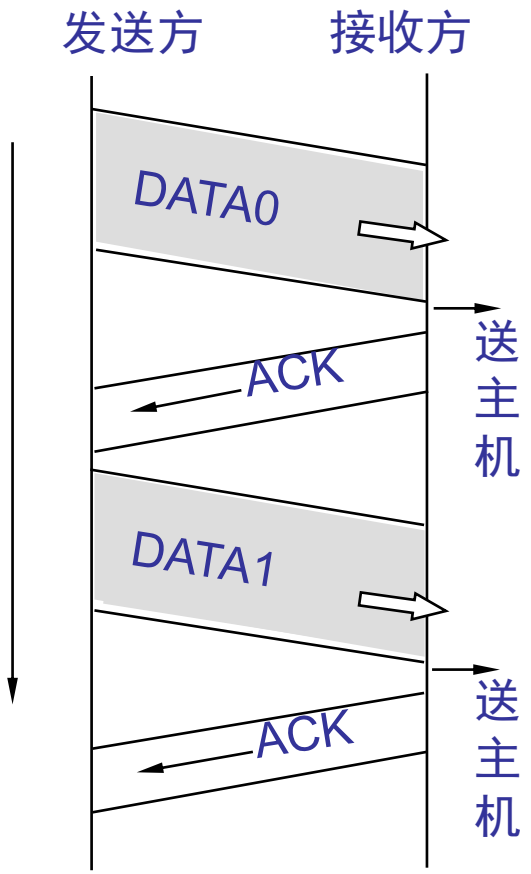
rdt 2.1：在具有比特差错信道上的有NAK的可靠数据传输协议

- ACK和NAK损坏怎么办?
  - 发送方不知道接收方的情况
  - 不能只重传：**重复接收**
- 重复接收怎么办?
  - 发送方**重发**——若收到被破坏的ACK或者NAK
  - 为每一个数据报加上**序号**
  - 接收方收到**重复序号**的数据则**丢弃**

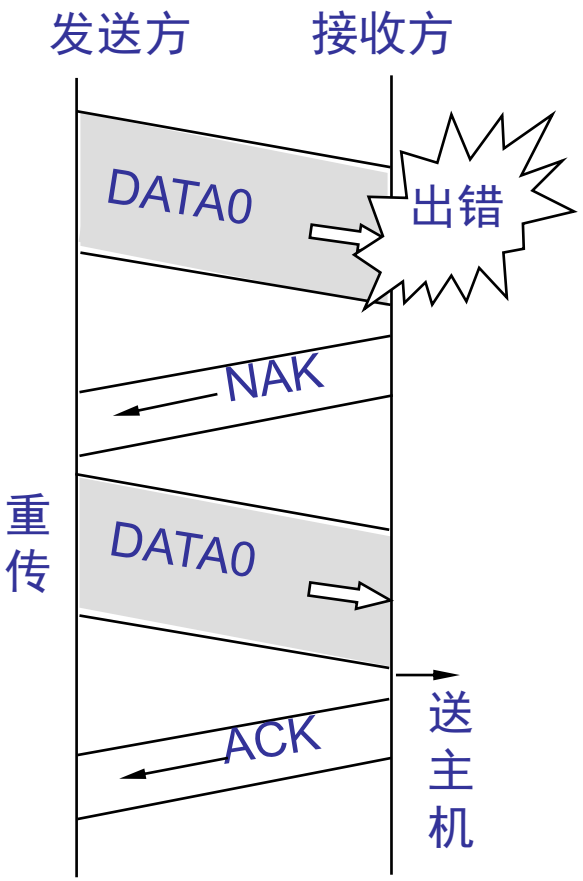
- rdt 2.1的发送方


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
等待来自
上层的
调用0
等待
ACK 或
NAK 0
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt)
Λ
等待
ACK 或
NAK 1
等待来自
上层的
调用1
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isNAK(rcvpkt))
udt_send(sndpkt)
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

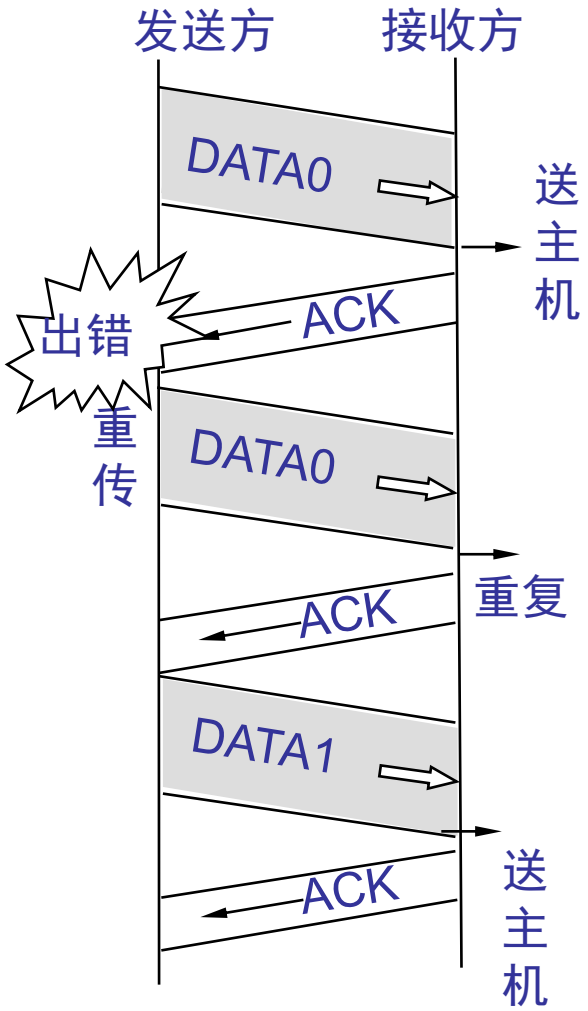
rdt2.1 时序图



(a) 正常情况

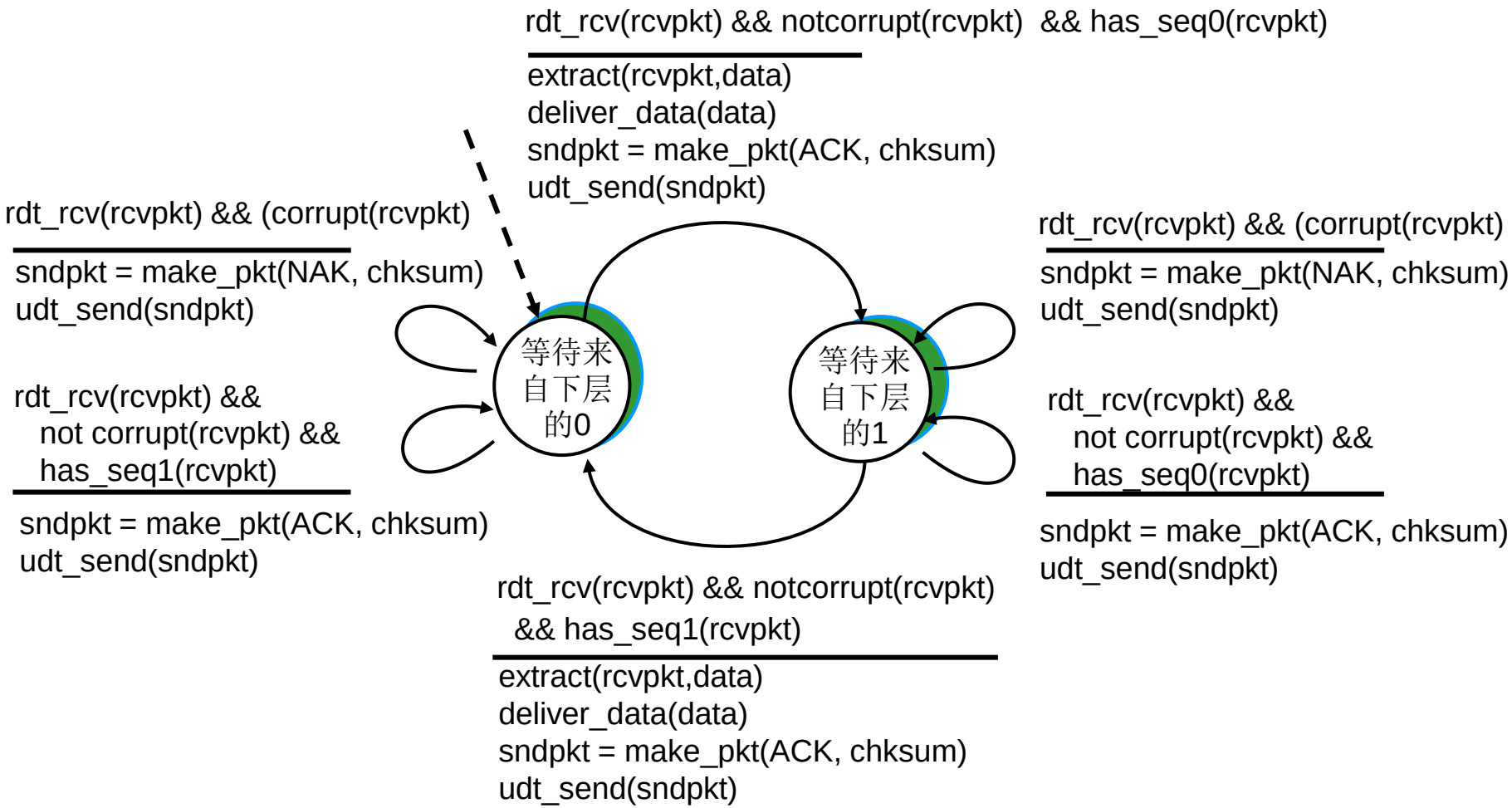


(b) 数据出错



(c) ACK/NAK出错

- rdt 2.1的接收方

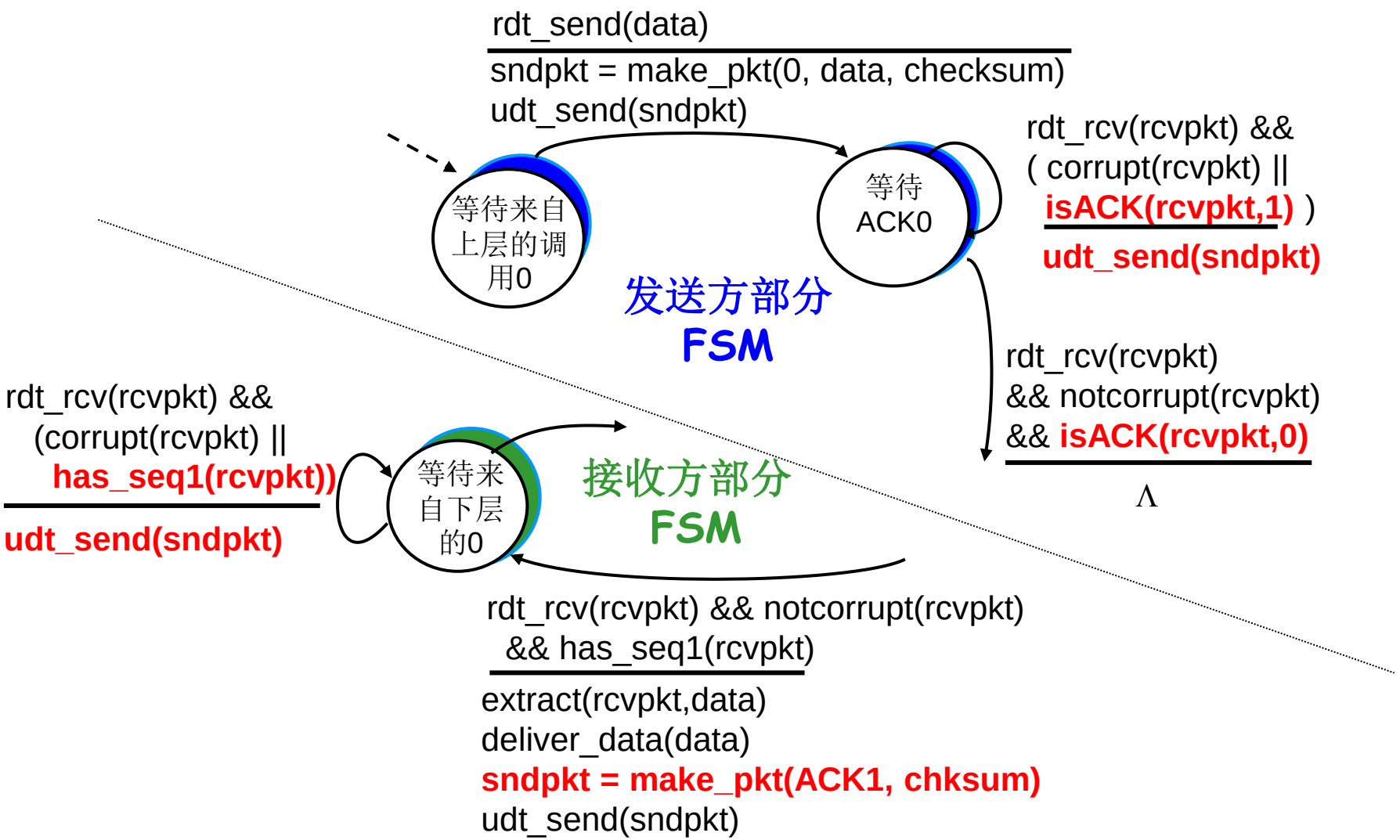

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
等待来自下层的0
等待来自下层的1
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

第三个版本——rdt 2.2

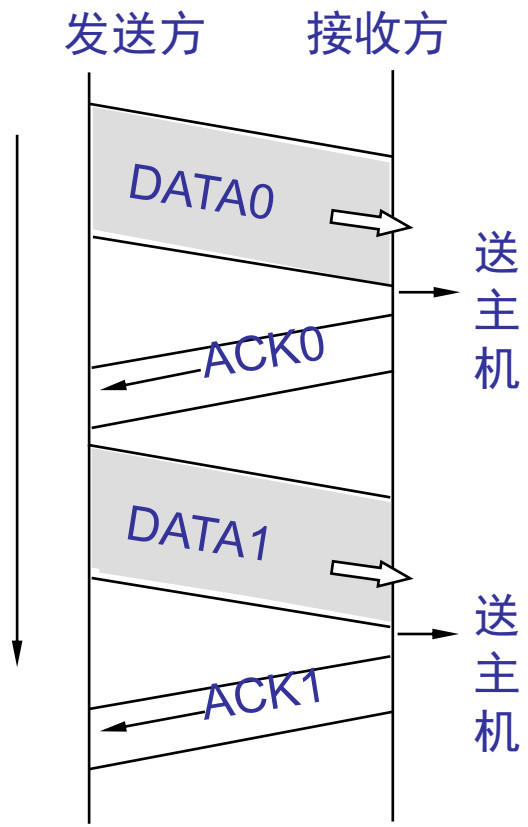
- 针对rdt 2.1的改进
  - **只**使用**ACK**
    - 反正“听”不清，总是要重传
    - 除了“**对**”的，都是“**错**”的
  - 如何通告“错”在哪里
    - 取消NAK，接收方**对最后一个正确收到的分组**发送 **ACK**
      - 接收方必须明确指出被确认的分组的**序号**
  - 发送方收到的**重复的ACK**将按照NAK来进行处理
    - **重传**正确的分组

停等协议：发送方发送数据报之后**等待**接收方的响应

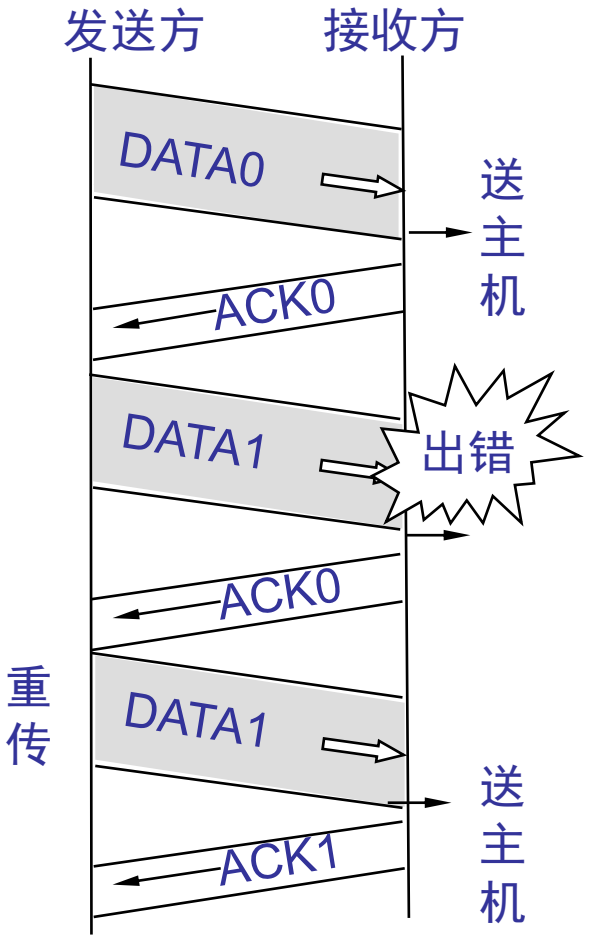
3.4 可靠数据传输的原理



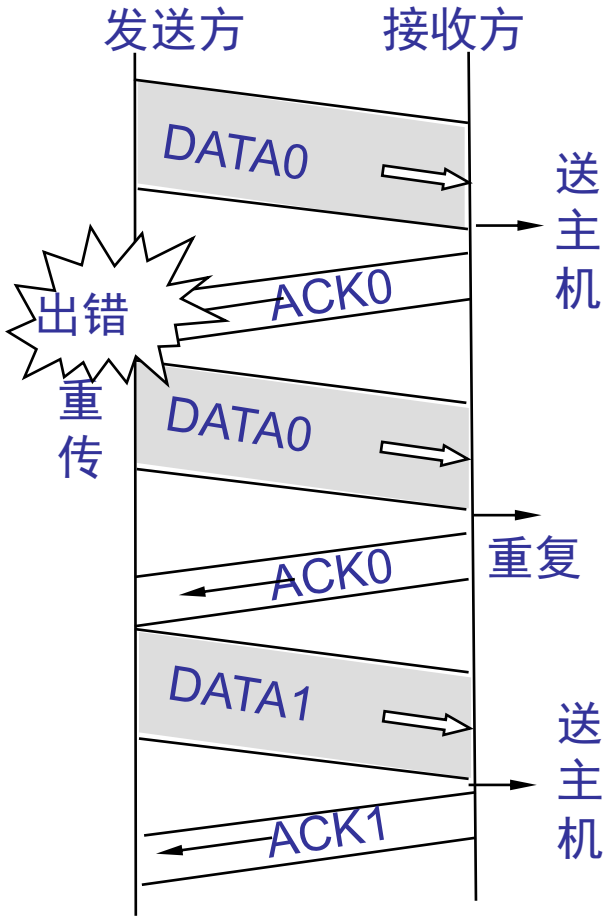
rdt2.2 时序图



(a) 正常情况



(b) 数据出错



(c) ACK出错

针对rdt 2.x的进一步讨论

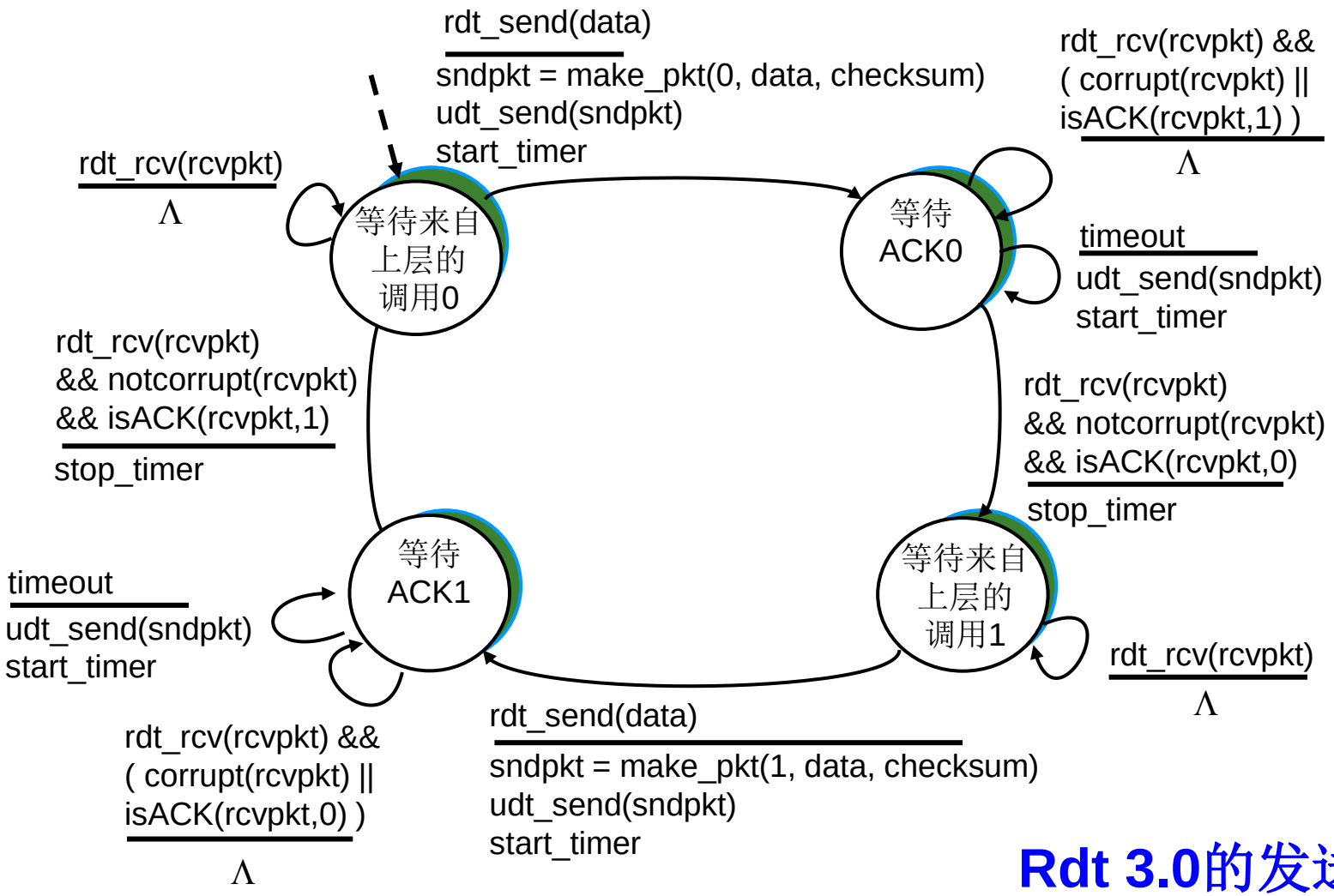
- rdt 2.x实际上也解决了传说中的流控问题

◆信道不但出错，而且丢包时——rdt 3.0

- ◆假设
  - ◆ 底层信道不但可能出现比特差错，而且可能会**丢包**（data,ACK）
  - ◆ 发生丢包后，如何处理
    - ◆ 校验和技术、序号、ACK、重传

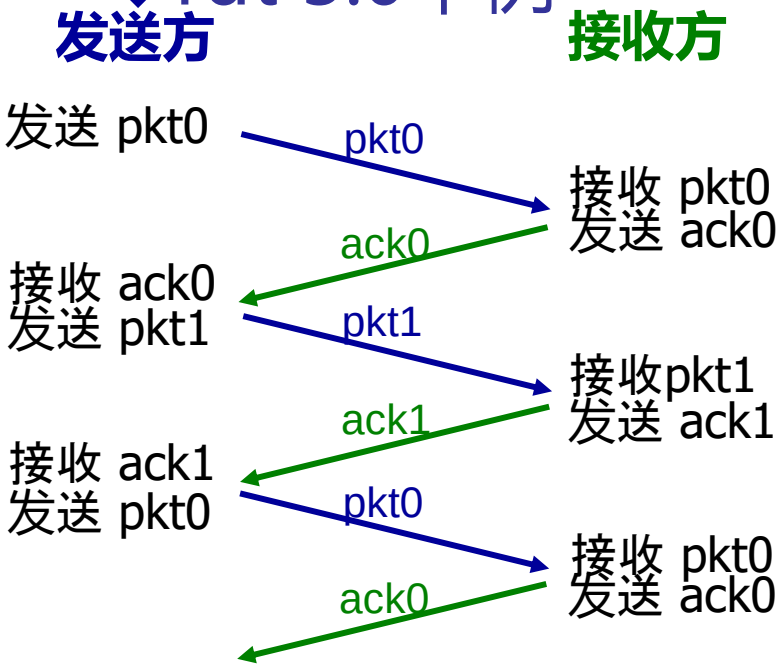
如何判断数据报丢失了呢？

最简单的方法就是：**耐心的等待！**

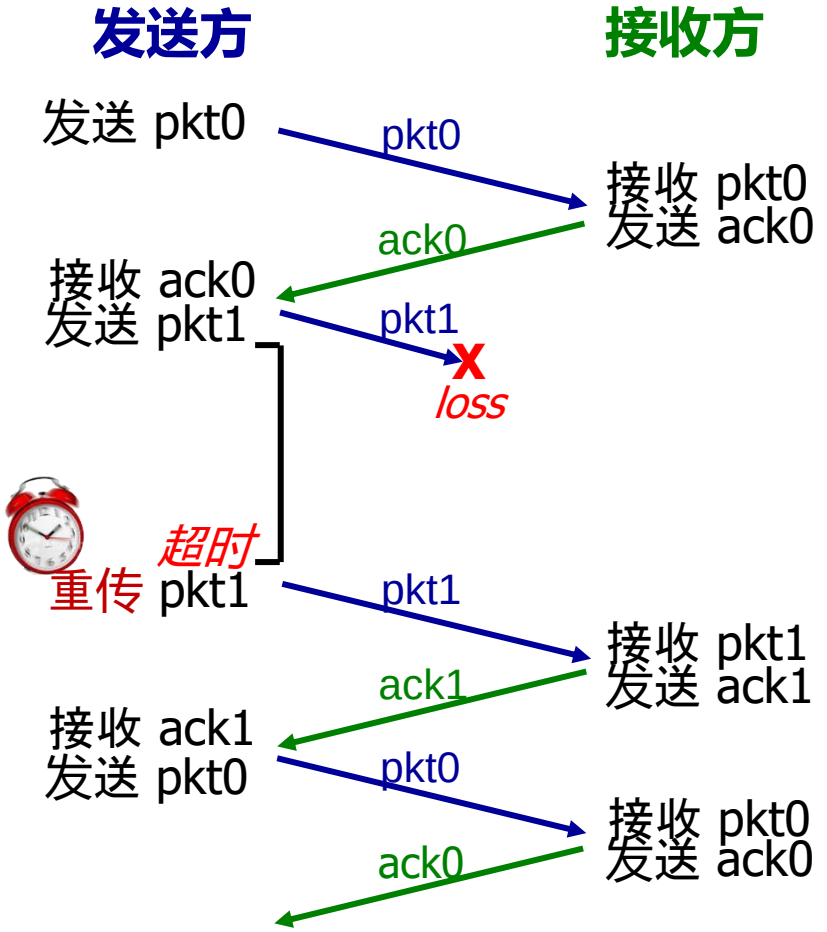

rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,1))
Λ
rdt_rcv(rcvpkt)
Λ
等待来自
上层的
调用0
等待
ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer
rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer
等待
ACK1
等待来自
上层的
调用1
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt)
Λ
rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) ||
isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

Rdt 3.0的发送方

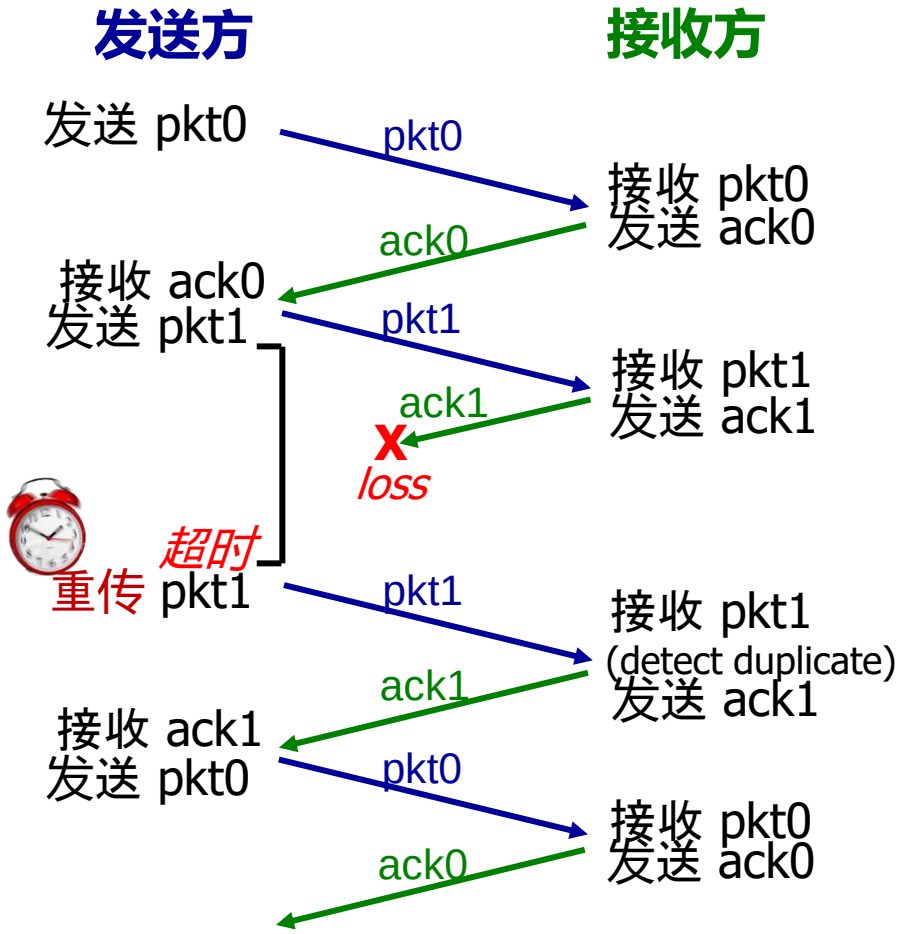
◆rdt 3.0举例



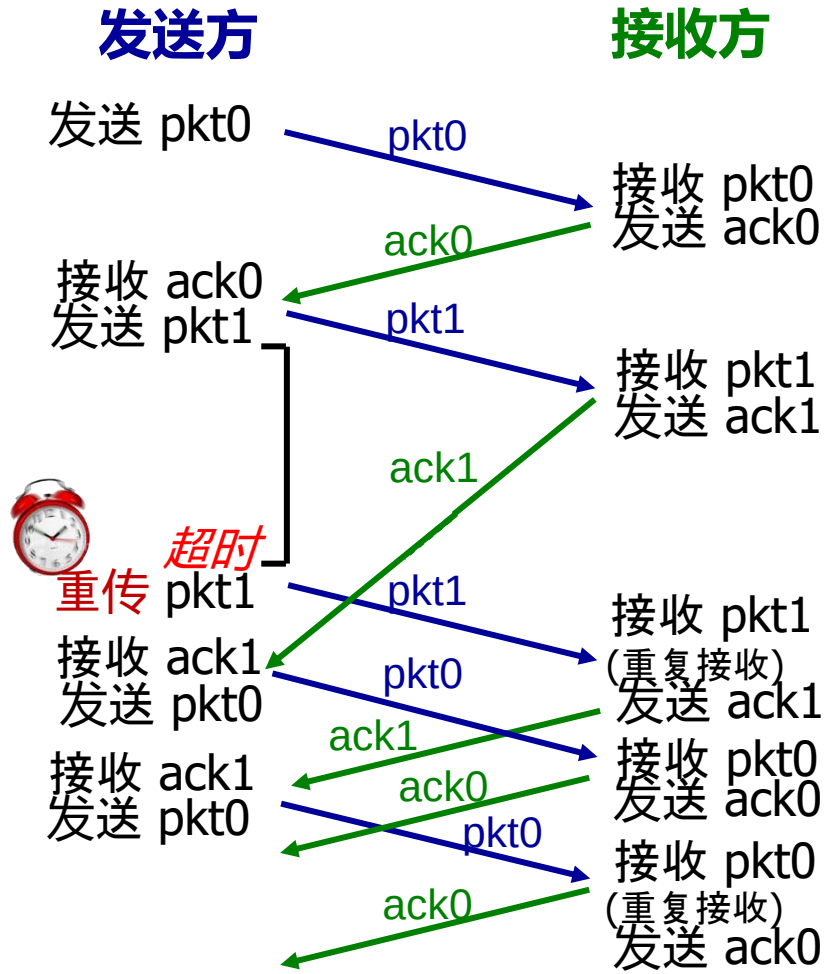
(a) 没有包丢失



(b) 包丢失



(c) ACK 丢失



(d) ACK 超时

◆rdt 3.0的性能分析

◆1Gbps 的链路, 15ms 的端到端延迟, 分组大小为1KB

$$T_{transmit} = \frac{L\ (比特为单位的分组大小)}{R\ (传输速率, bps)} = \frac{8kb/pkt}{10^9\ b/sec} = 8\ \mu s$$

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

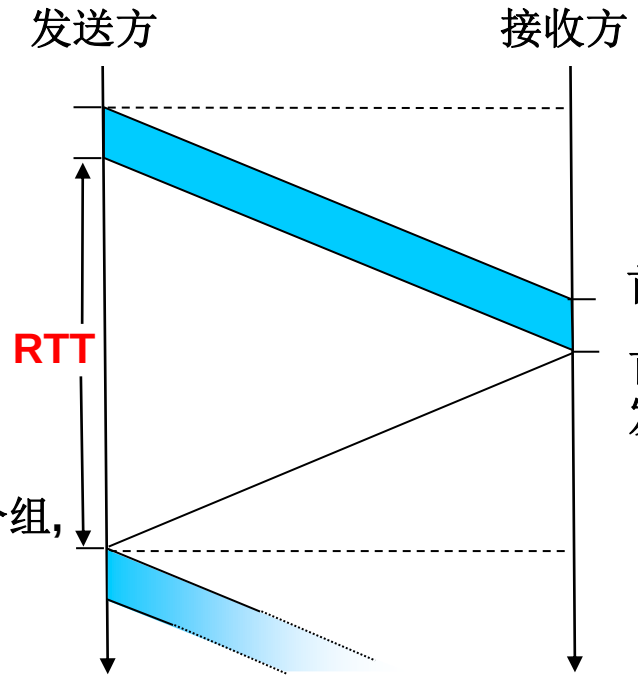
- 每30ms（2*15ms 往返时延）内只能发送1KB： 1 Gbps的链路只有33kB/sec 的吞吐量
- 网络协议限制了物理资源的利用率!

◆rdt 3.0性能低下的原因

◆停等操作

首个分组的第1个比特被传输，
t = 0

首个分组的最后1比特被传输，
t = L / R



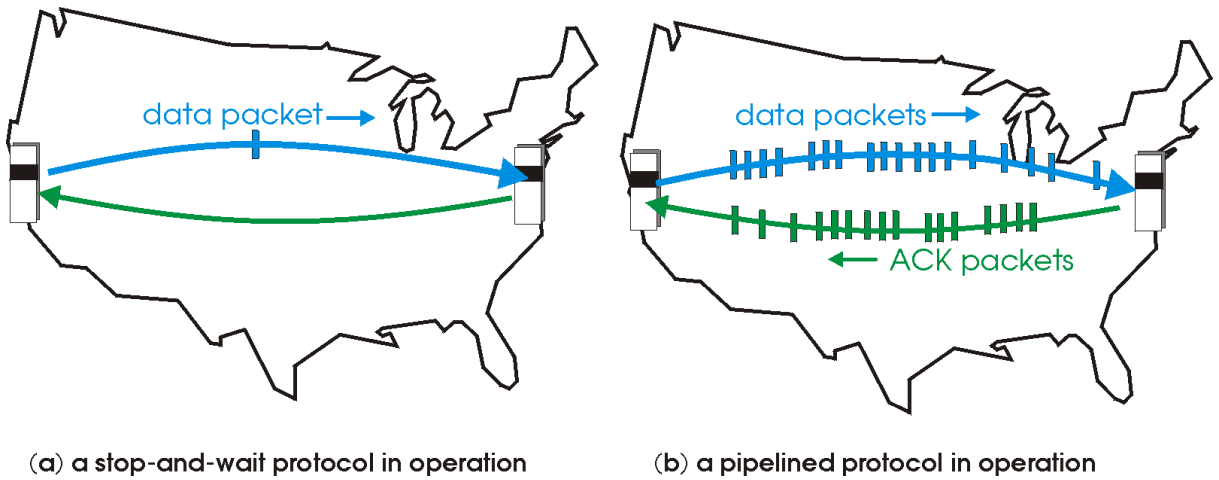
首个分组的第1个比特到达

首个分组的最后1个比特到达，
发送ACK

ACK 到达，发送下一个分组，
t = RTT + L / R

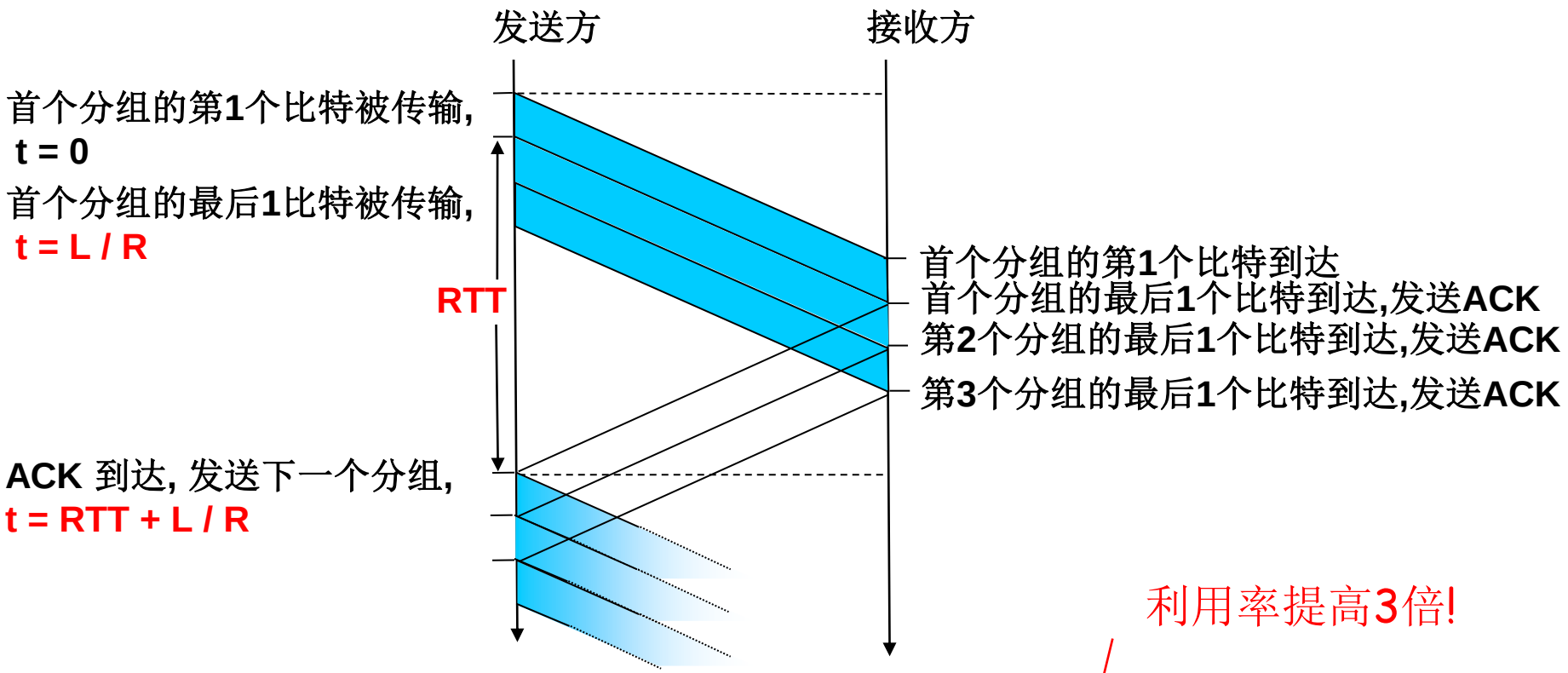
$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

- 提高性能的方法：流水线技术
  - 允许**发送方**发送**多个**分组而**无需等待确认**
    - **原本的两个序号（0，1）够用吗?**
      - **必须增大序号范围**
    - **发送方需要缓存吗?**
      - **发送方**最低限度应当**能缓存**那些已发送但未被确认的**分组**
    - **接收方需要缓存吗?**
      - 或许需要缓存那些已经正确接收的分组
    - **缓存多少?**
      - **和序号长度有关**
    - **能够不间断的发送吗?**
      - **不能? why?**



(a) a stop-and-wait protocol in operation　　(b) a pipelined protocol in operation

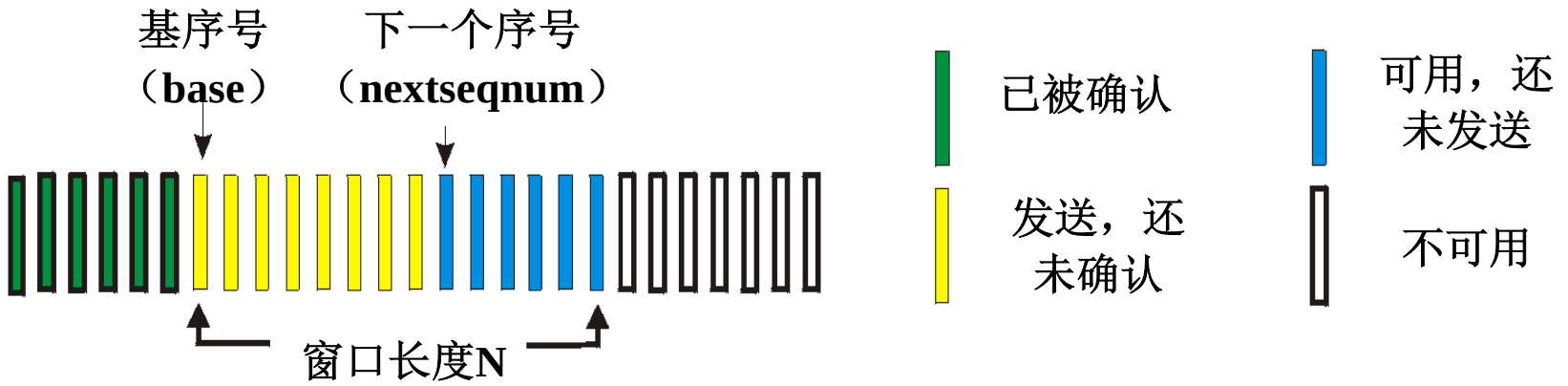
•流水线技术对性能提升的原理图



$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

利用率提高3倍!

- 流水线技术工作原理
  - 分组首部用k-比特字段表示序号
  - **未被传输**和**已被传输但还未确认**的分组的许可**序号范围**可以看作是一个在序号范围内大小为N的“**窗口**(window)”
    - 窗口——序号集合、序号管理器



- 问题：当流水线技术中丢失一个分组后，如何进行重传
  - Go-Back-N（GBN）协议：其后分组全部重传
    - 思考：为什么要回退N?
  - 选择重传（SR）协议：仅重传该分组
    - 思考：为什么可以选择?

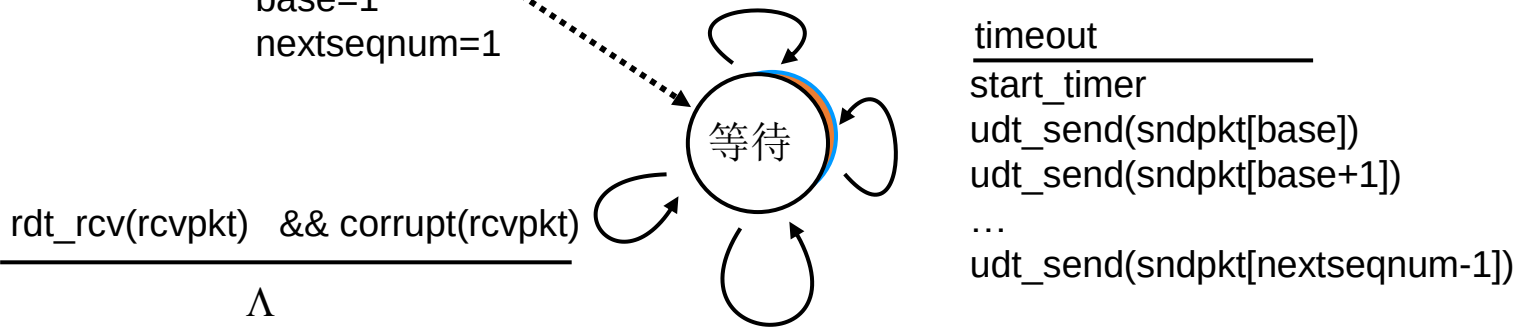
发送方的FSM图

rdt_send(data)

```
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
       start_timer
    nextseqnum++
    }
else
  refuse_data(data)
```

Λ

base=1
nextseqnum=1



timeout

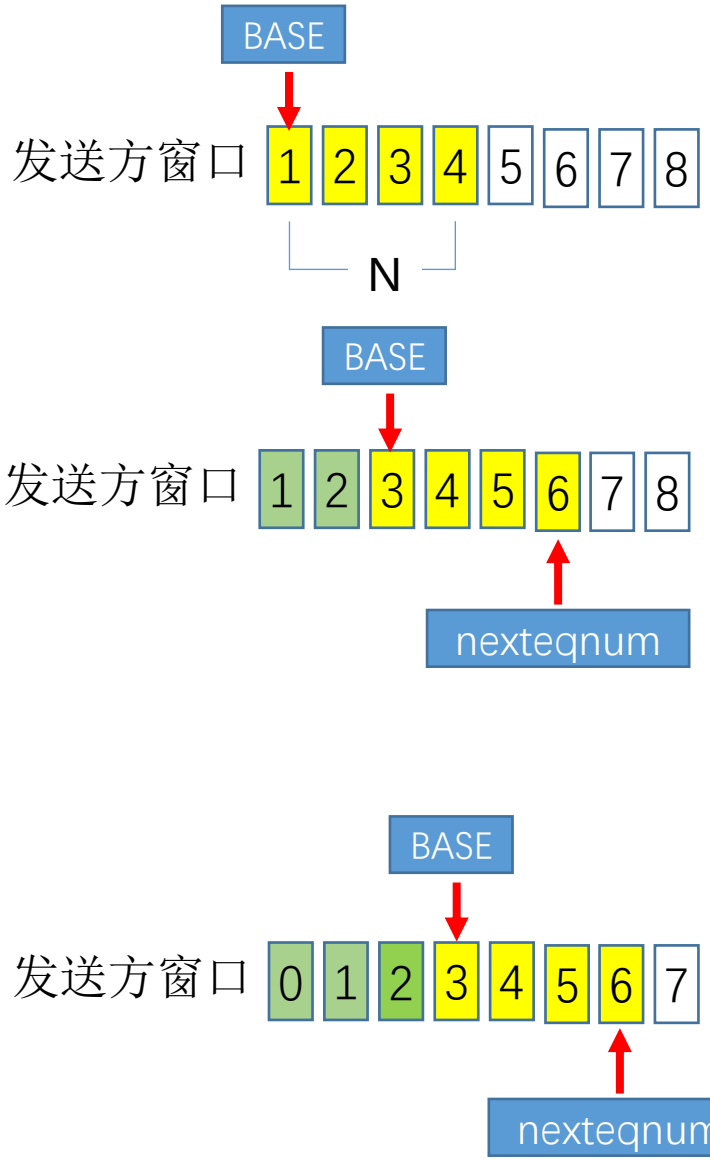
start_timer
udt_send(sndpkt[base])
udt_send(sndpkt[base+1])
…
udt_send(sndpkt[nextseqnum-1])

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

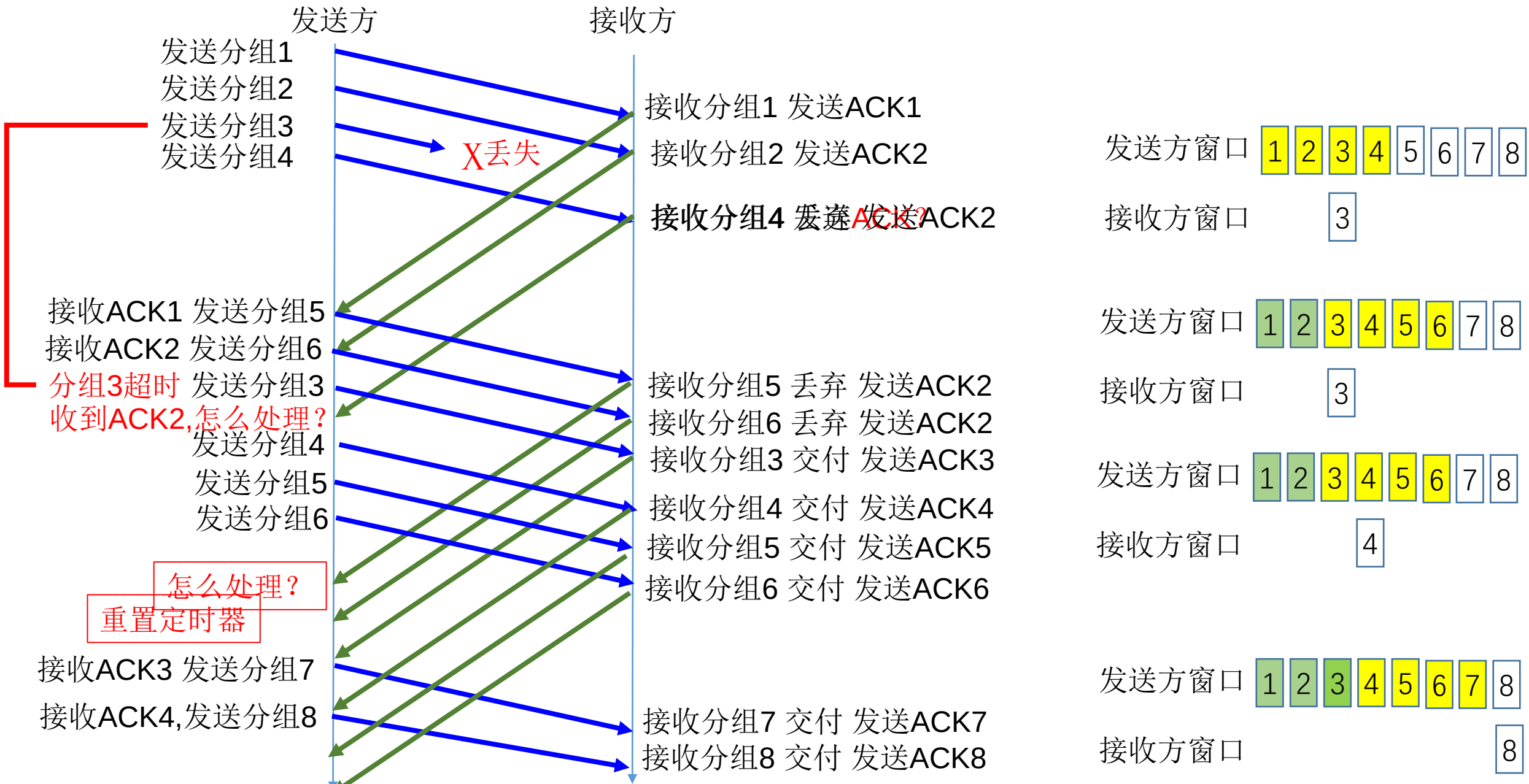
Λ

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

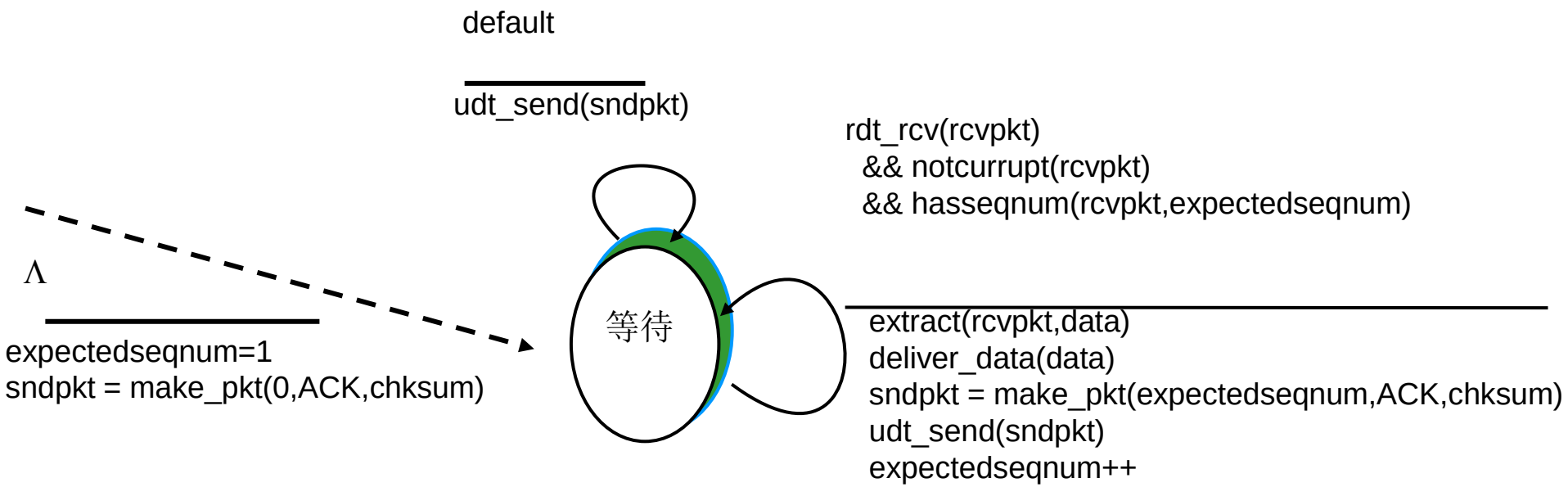
```
base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
  else
    start_timer
```



GBN时序图


发送方
接收方
发送分组1
发送分组2
发送分组3
发送分组4
X丢失
接收分组1 发送ACK1
接收分组2 发送ACK2
接收分组4 丢弃 发送ACK2
发送方窗口 1 2 3 4 5 6 7 8
接收方窗口 3
接收ACK1 发送分组5
接收ACK2 发送分组6
分组3超时 发送分组3
收到ACK2,怎么处理？
发送分组4
发送分组5
发送分组6
接收分组5 丢弃 发送ACK2
接收分组6 丢弃 发送ACK2
接收分组3 交付 发送ACK3
接收分组4 交付 发送ACK4
接收分组5 交付 发送ACK5
接收分组6 交付 发送ACK6
发送方窗口 1 2 3 4 5 6 7 8
接收方窗口 3
发送方窗口 1 2 3 4 5 6 7 8
接收方窗口 4
怎么处理？
重置定时器
接收ACK3 发送分组7
接收ACK4,发送分组8
接收分组7 交付 发送ACK7
接收分组8 交付 发送ACK8
发送方窗口 1 2 3 4 5 6 7 8
接收方窗口 8

◆Go-Back-N协议



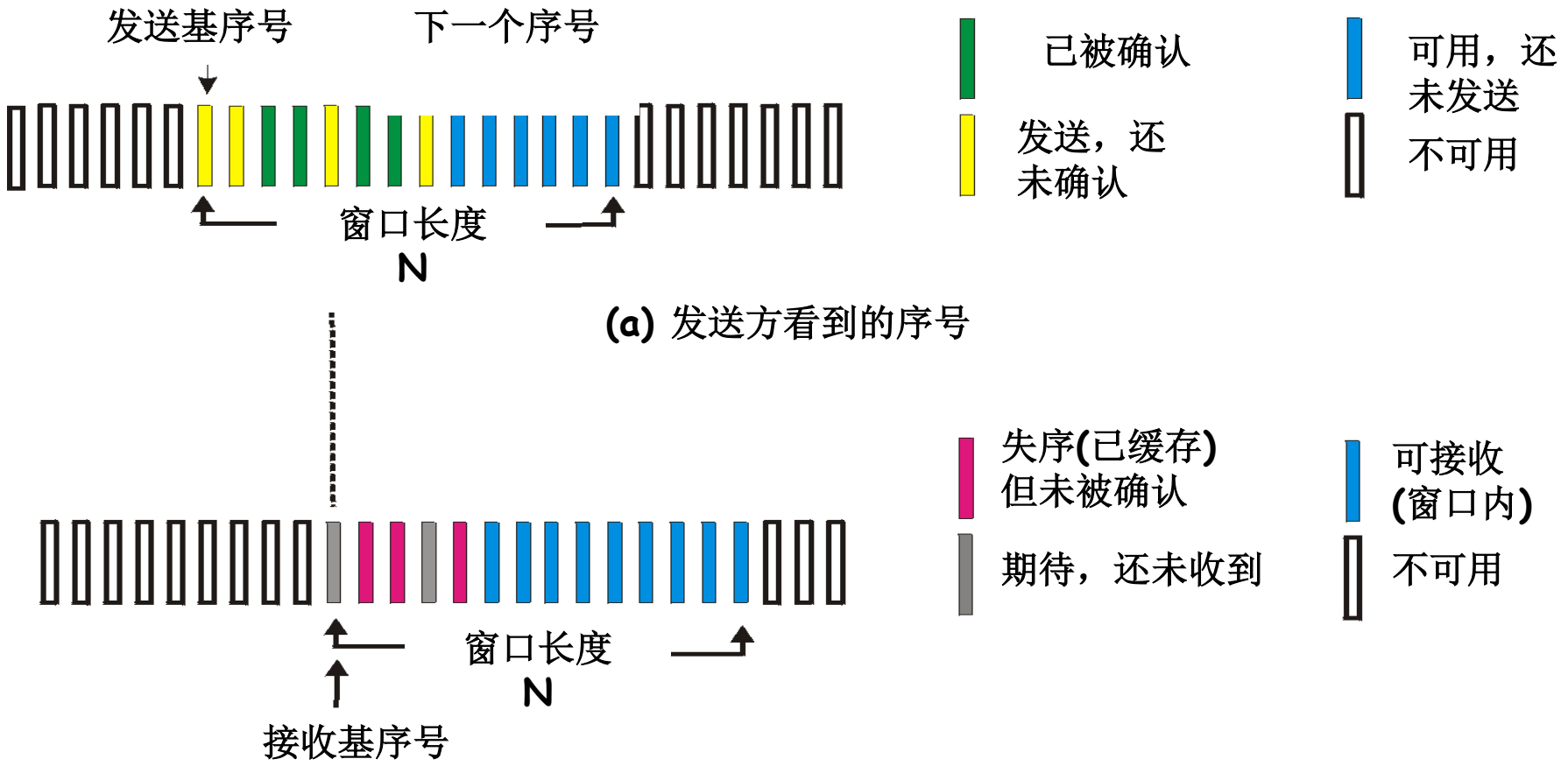
接收方的FSM图

- **Go-Back-N协议**
  - 特点
    - ACK(n): 接收方对序号n之前包括n在内的所有分组进行确认 - “**累积 ACK**”
    - 允许发送方能且只能连续发送n个数据包，若窗口大小为n，同时，窗口中**未被确认的分组数不能超过n**。
    - 对所有已发送但未确认的分组**统一**设置一个**定时器**，从一次流水的最“老”分组开始计时
    - ***超时(n):*** 重传分组n和窗口中所有序号大于n的分组
  - 思考
    - 收到重复的分组怎么处理？ （出现超时重传时）
      - 丢弃分组，重发ACK
    - 分组失序怎么办？ （有分组丢失，后面的分组正确到达）
      - 丢弃 (不缓存) -> 接收方无缓存!
      - 重发按序到达的最高序号分组的ACK（重发该序号以后的分组）

3.4 可靠数据传输的原理

- Go-Back-N的滑动窗口大小
  - 发送端 ≤ 2^k-1
  - 接收端 = 1

◆选择重传（SR）协议


发送基序号
下一个序号
窗口长度
N
已被确认
发送，还未确认
可用，还未发送
不可用
(a) 发送方看到的序号
失序(已缓存)但未被确认
期待，还未收到
可接收(窗口内)
不可用
窗口长度
N
接收基序号
(b) 接收方看到的序号

◆选择重传（SR）协议

◆发送方

- ◆ 从上层收到数据
  - ◆ 如果下一个可用于该分组的序号在窗口内，则将数据打包并发送
- ◆ 超时（n）
  - ◆ 为每一个分组定义定时器
  - ◆ 重传分组n, 重置定时器
- ◆ 收到确认(n) 在 [sendbase,sendbase+N-1]范围内
  - ◆ 标记分组 n 为已接收
  - ◆ 如果n是发送窗口基序号sendbase，则将窗口基序号前推到下一个未确认序号

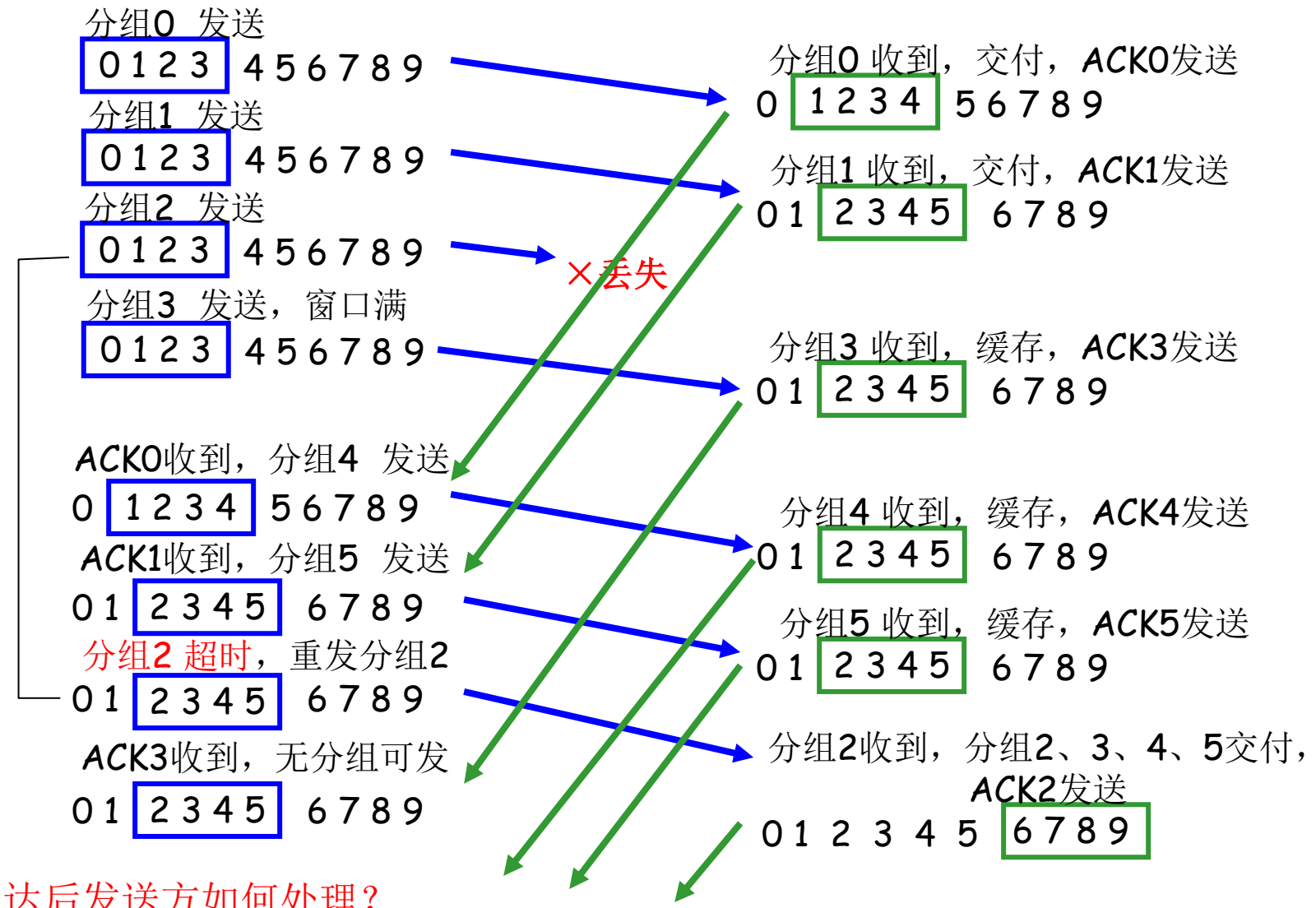
◆选择重传（SR）协议

◆接收方

- ◆ 分组序号n在[rcvbase, rcvbase+N-1]范围内
  - ◆ 发送n的确认ACK(n)
  - ◆ 如果分组序号不连续(失序)：将其缓存
  - ◆ 按序分组: 将该分组以及以前缓存的序号连续的分组一起交付给上层, 将窗口前推到下一个未收到的分组
- ◆ 分组序号n在[rcvbase-N, rcvbase-1]范围内
  - ◆ 虽然曾经确认过，仍再次发送n的确认ACK(n)
- ◆ 其他情况
  - ◆ 忽略该分组

3.4 可靠数据传输的原理

◆SR操作



Q：1）ACK2到达后发送方如何处理？

2）如果ACK2超时会发生什么情况？

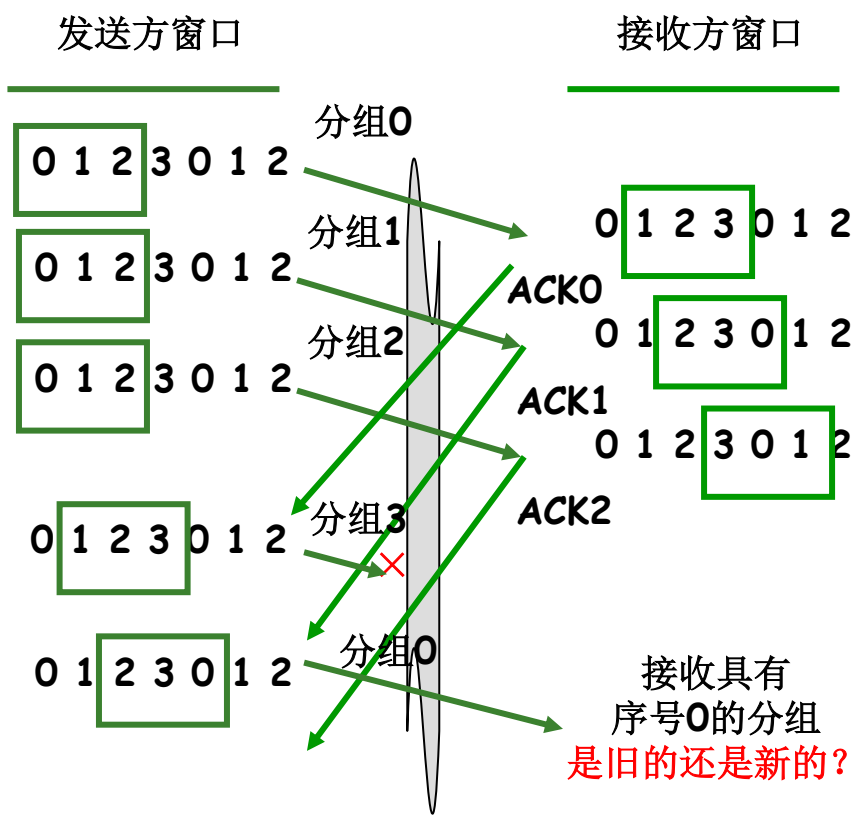
◆SR接收方窗口太大的困境

◆有限序号范围，缺乏同步

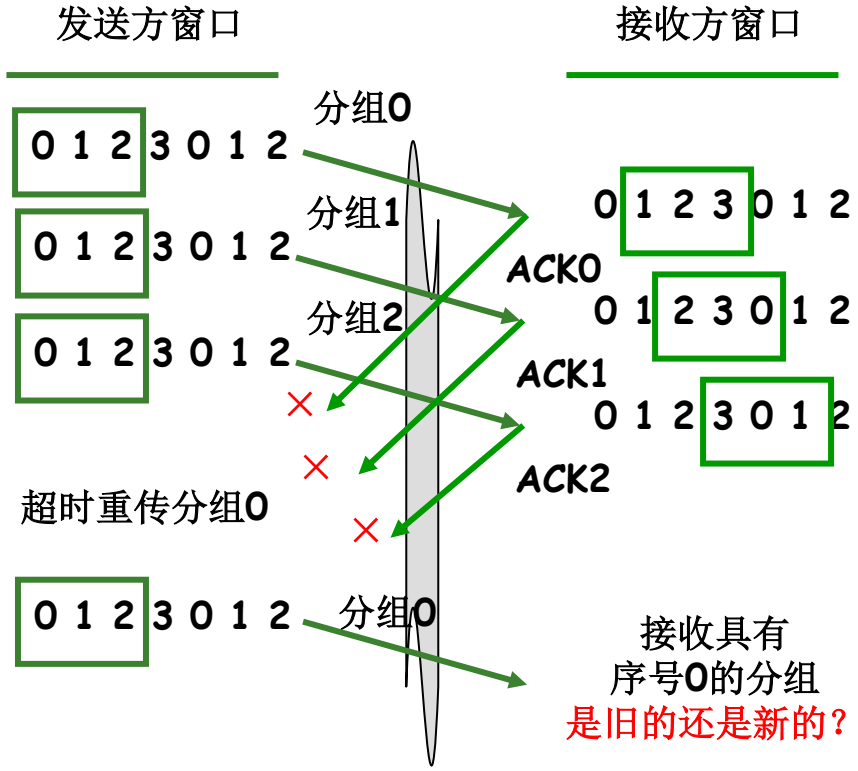
- ◆ 窗口必须小于序号范围(序号空间)

◆窗口必须多小?

结论：接收方窗口 ≤ 2^{k-1}

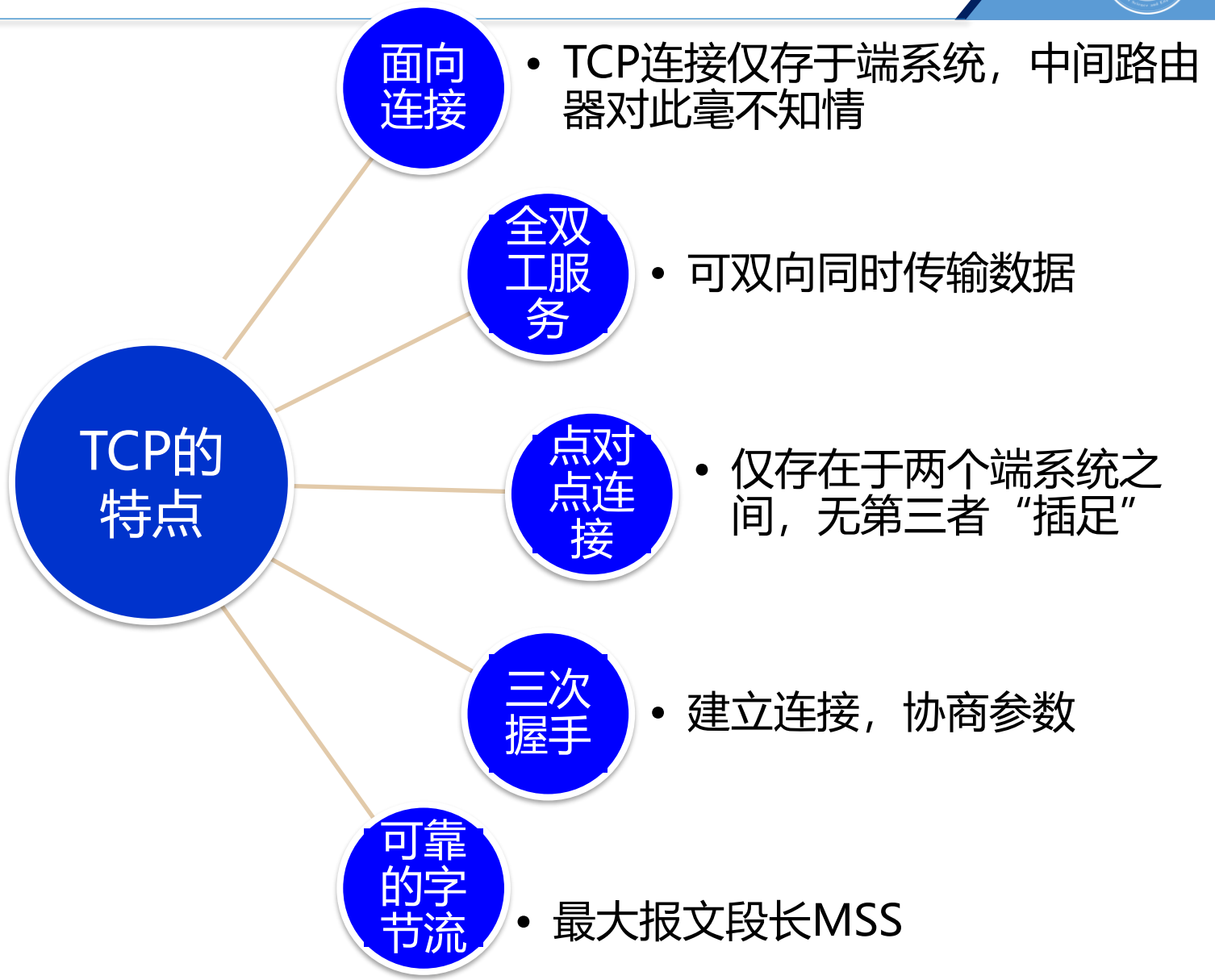


（a）正确



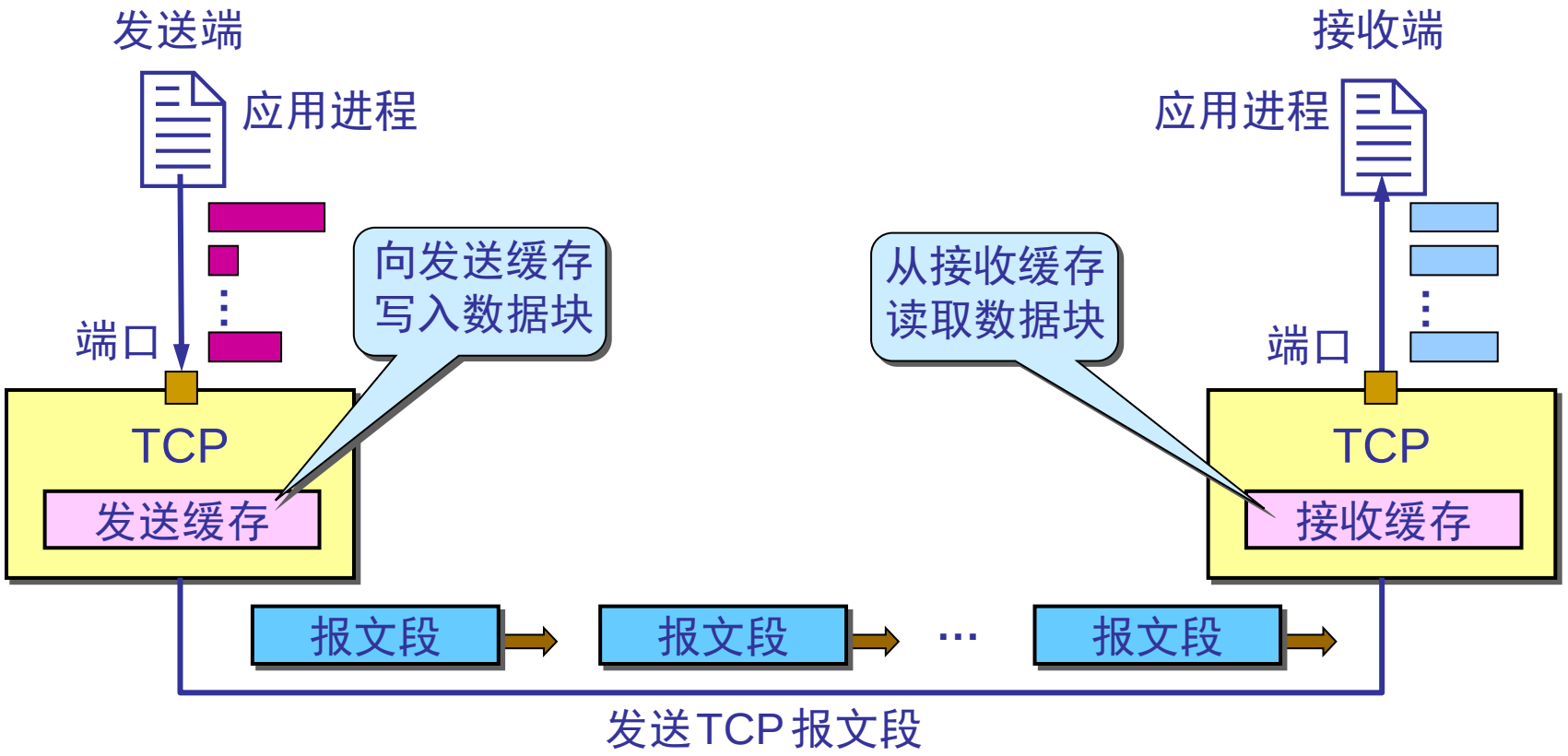
（b）无法判断

3.5 面向连接的传输: TCP



3.5 面向连接的传输: TCP

◆ TCP连接

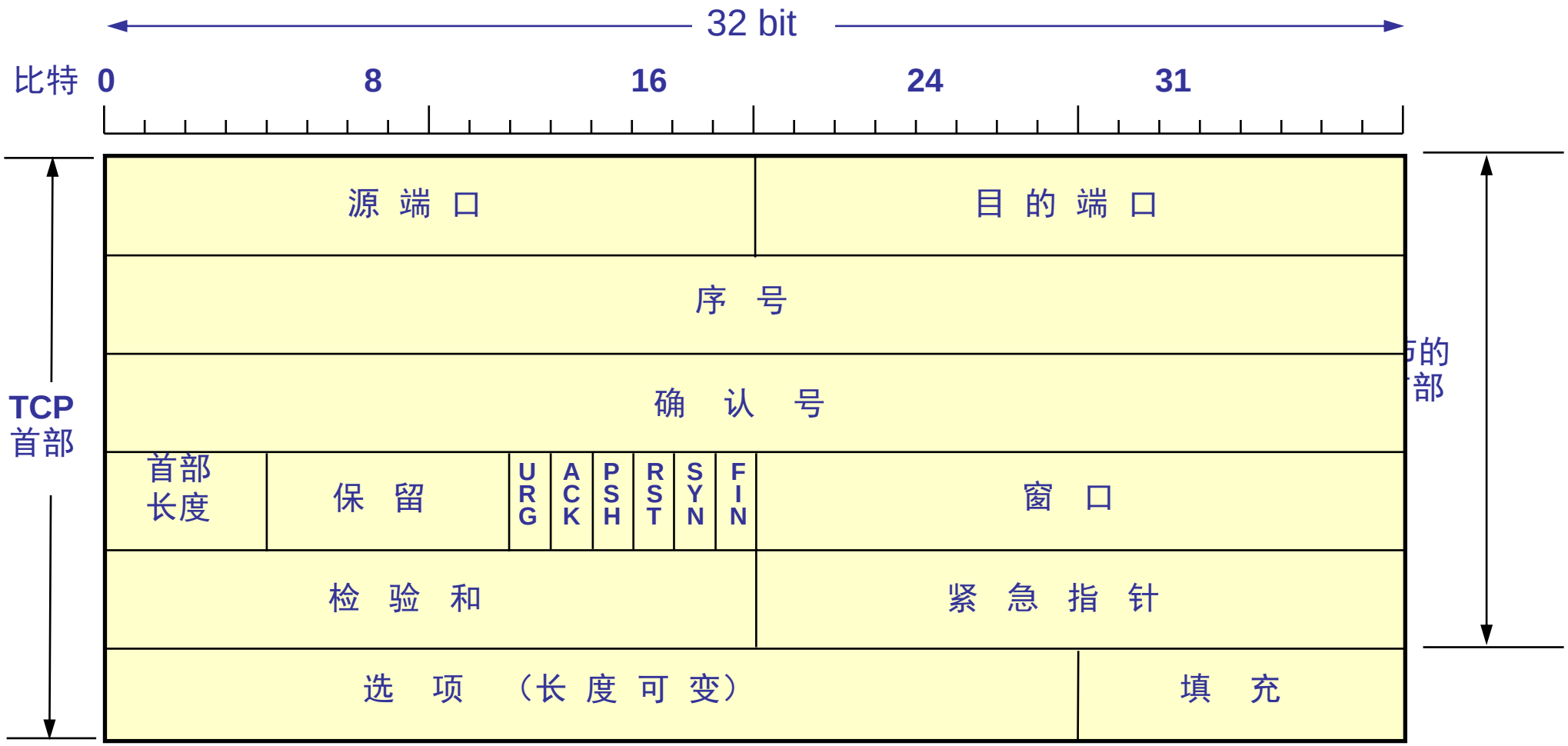


◆ MSS 最大报文段长度

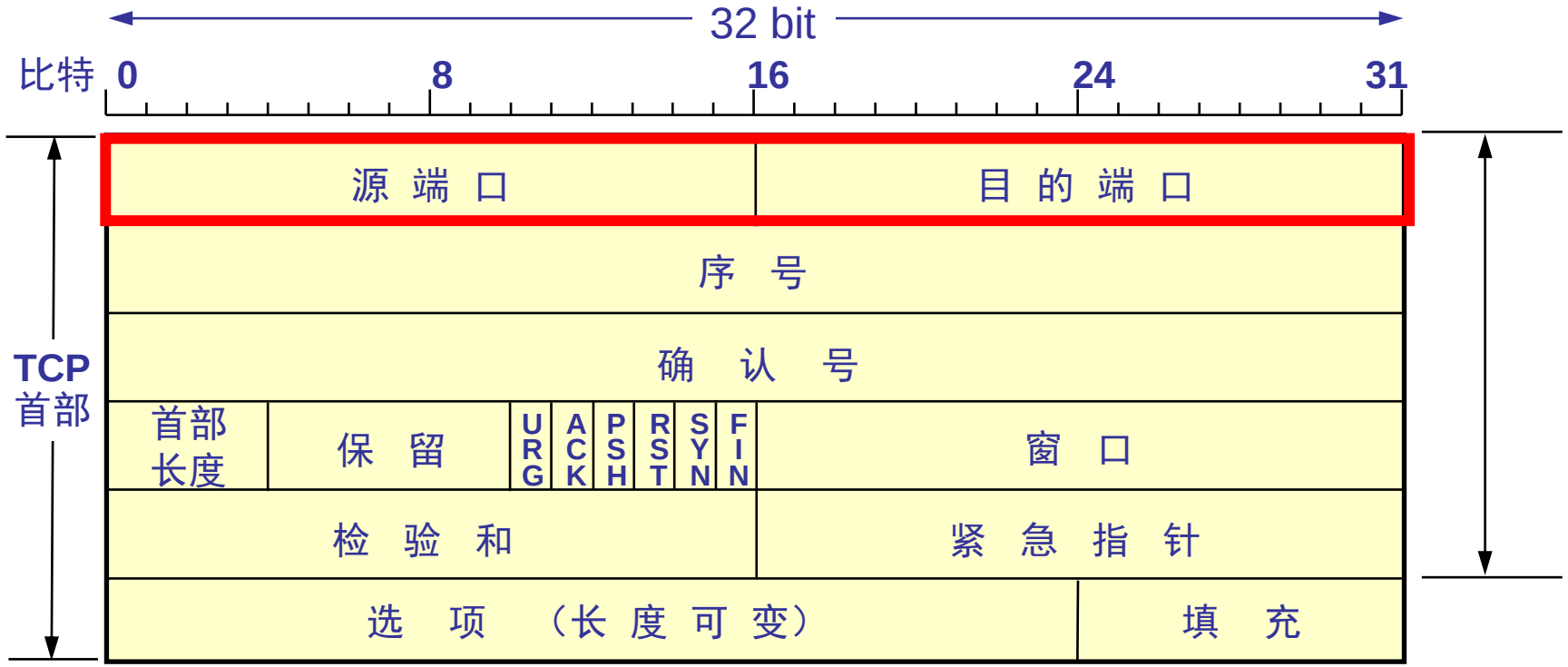
- ◆ 与MTU（最大传输单元的关系）
- ◆ 典型值=1460？ （以太网和PPP链路具有1500MTU-40链路层首部）

3.5 面向连接的传输: TCP

◆TCP报文段首部结构

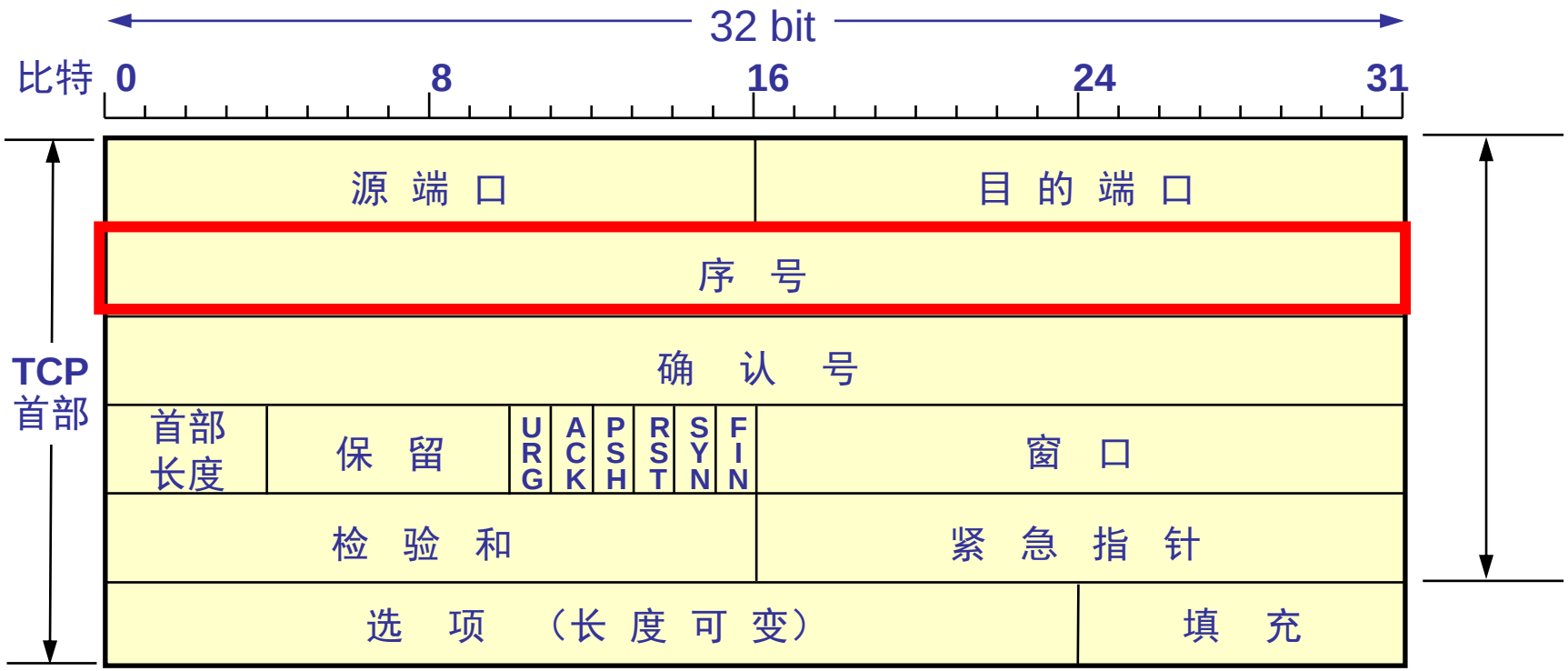


3.5 面向连接的传输: TCP



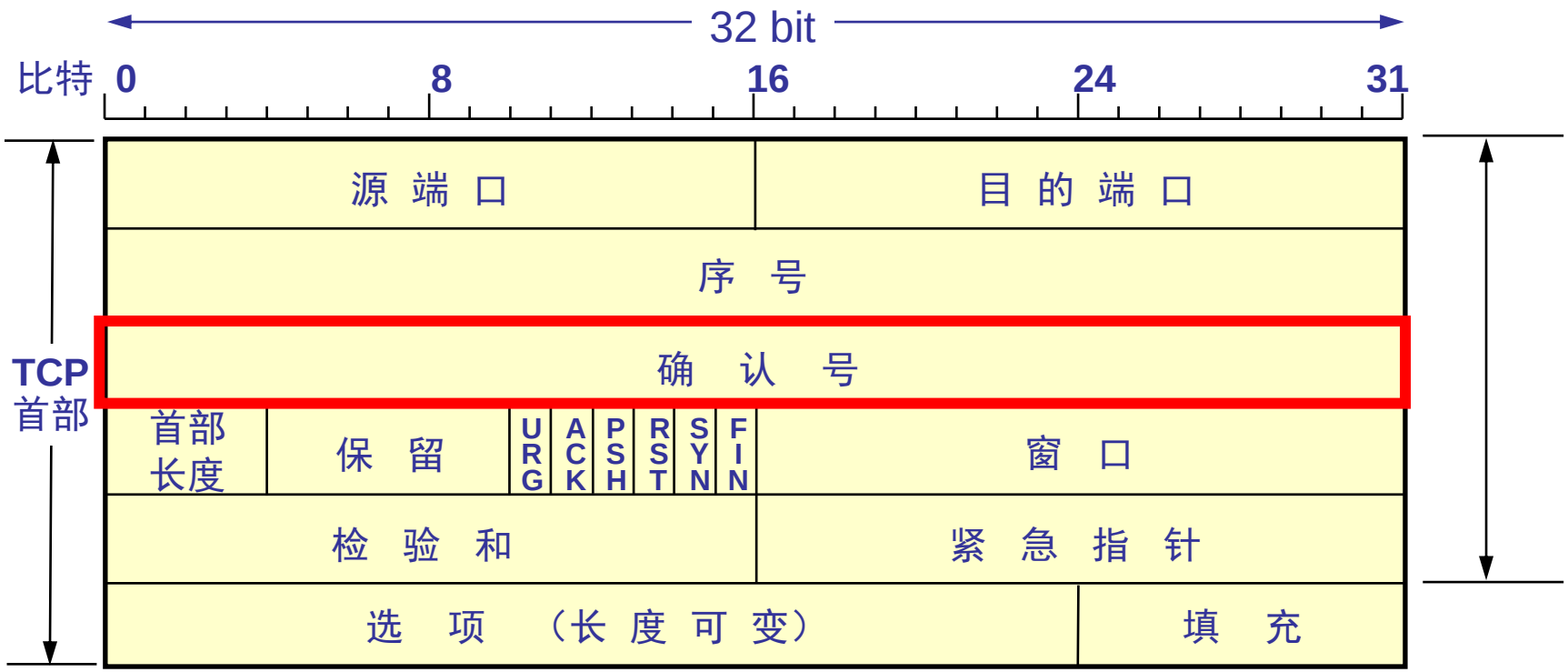
源端口和目的端口字段——各占 2 字节。端口是运输层与应用层的服务接口。运输层的复用和分用功能都要通过端口才能实现。

3.5 面向连接的传输: TCP



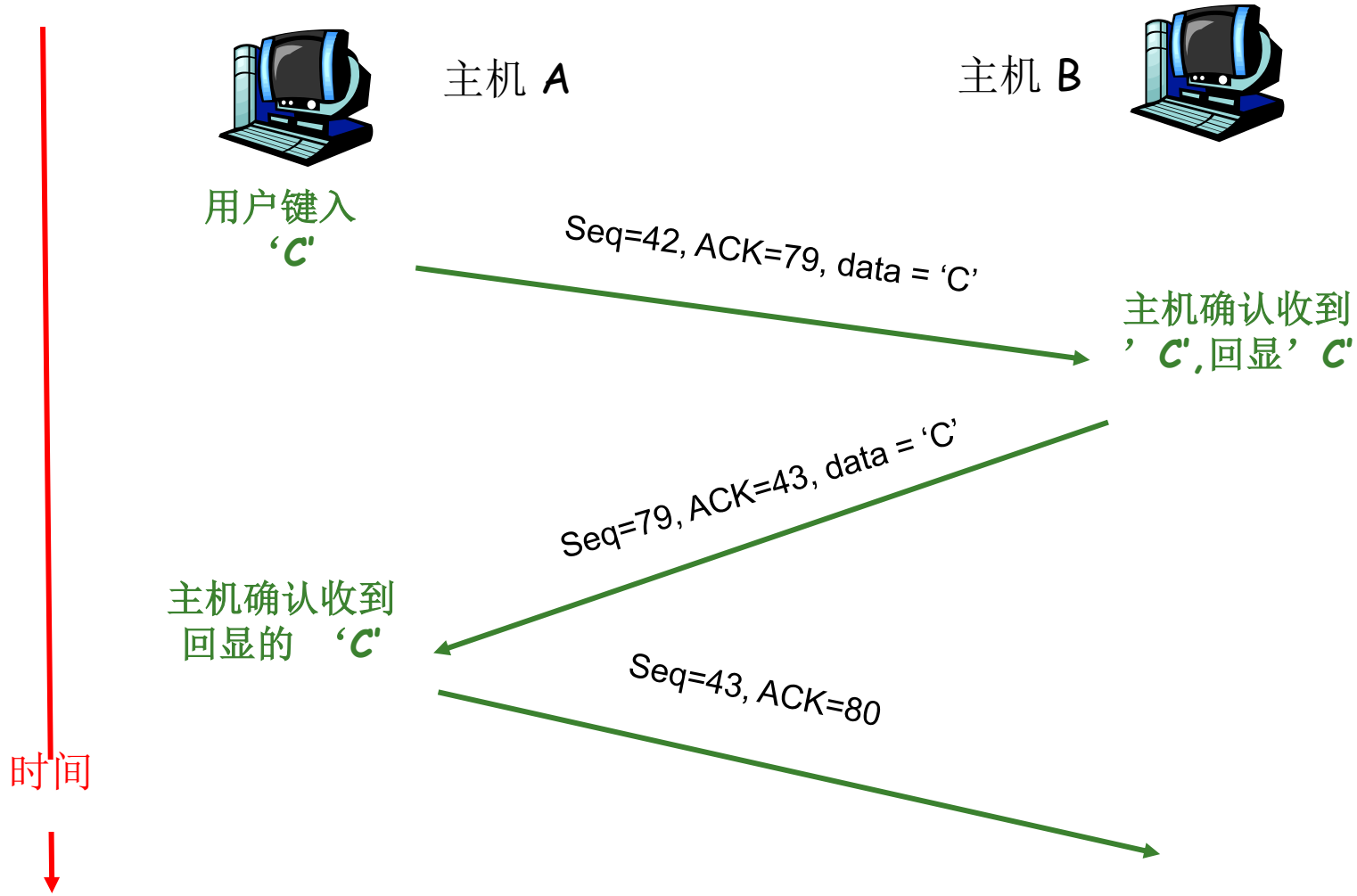
序号字段——占 4 字节。TCP 连接中传送的数据流中的**每一个字节**都编上**一个序号**。**序号字段**的值则指的是本报文段所发送的数据的**第一个字节**在整个报文字节流中的**序号**。

3.5 面向连接的传输: TCP



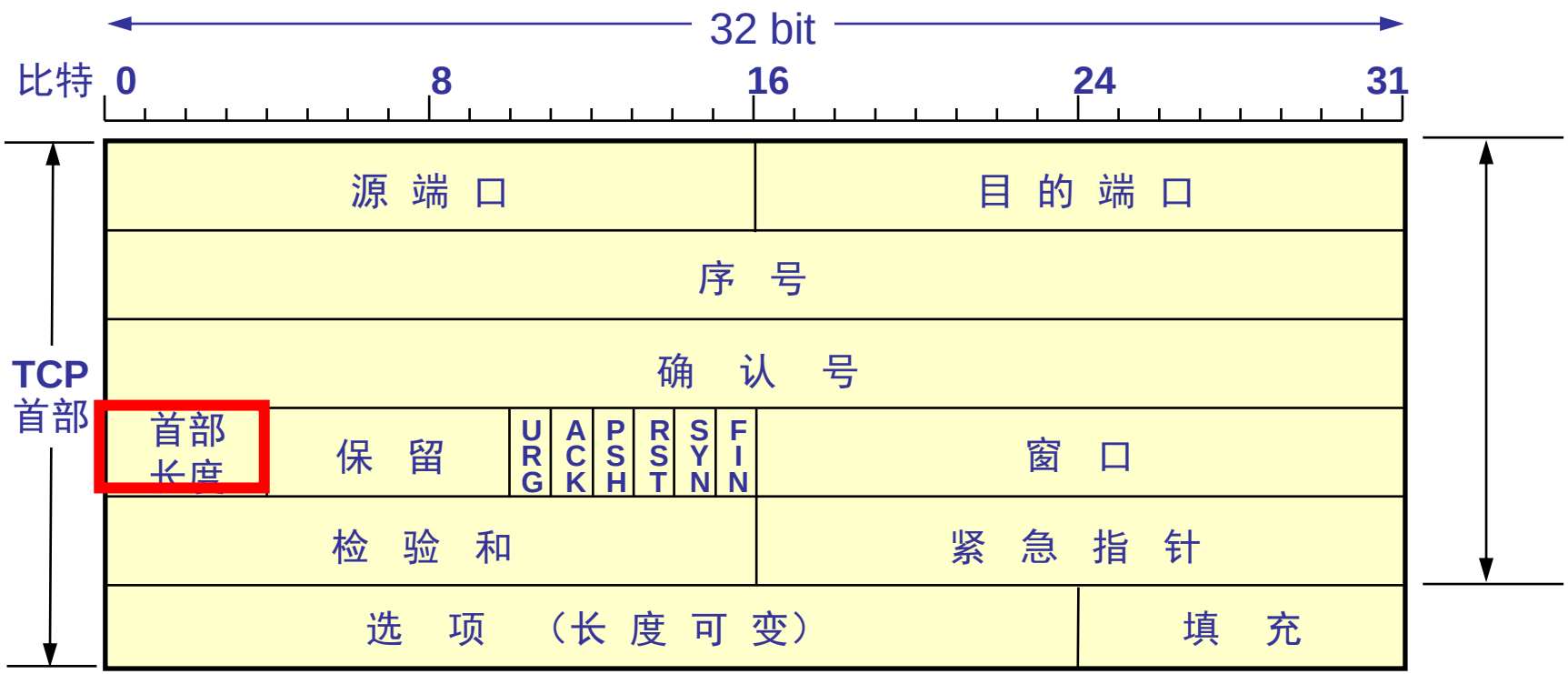
确认号字段——占 4 字节，是**期望收到**对方的下一个**报文段**的数据的第一个字节的**序号**。

◆TCP序列号和确认序列号



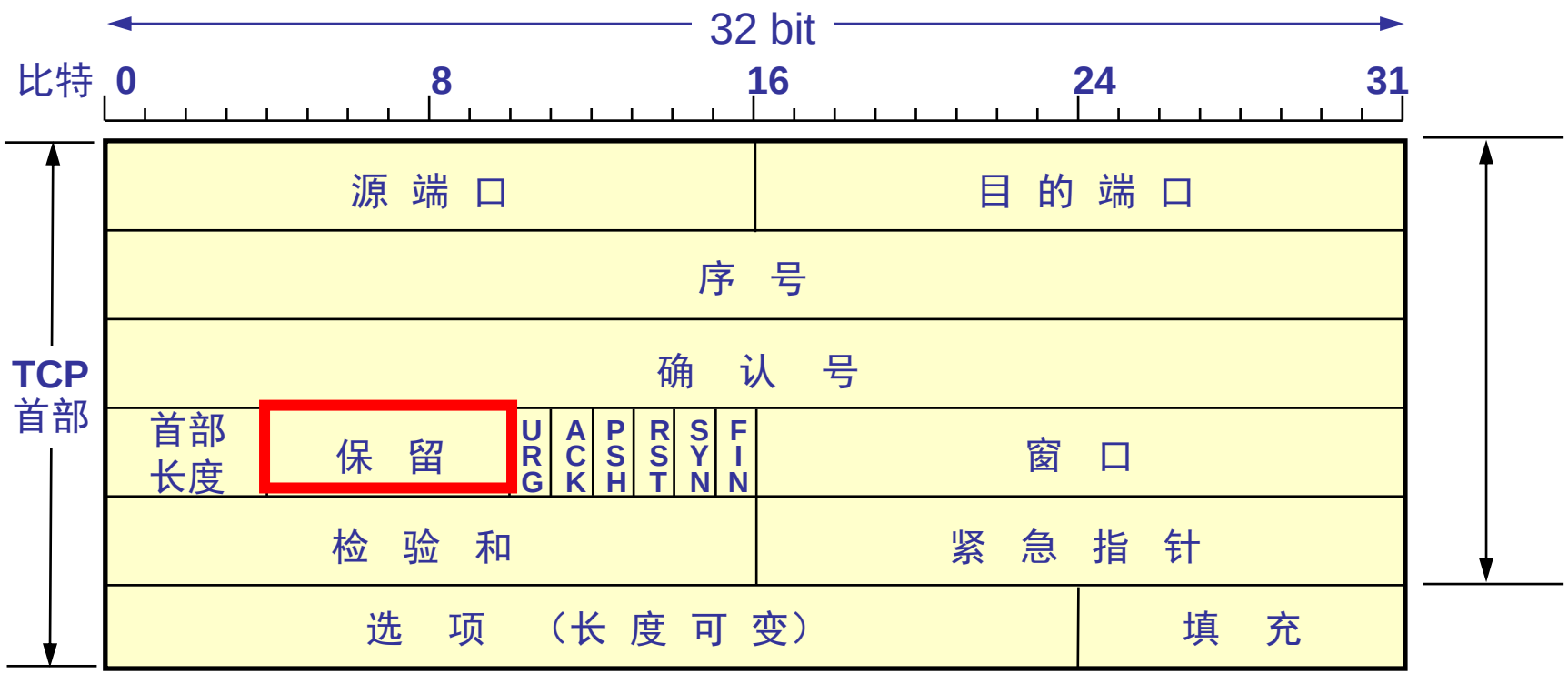
简单的Telnet例子

3.5 面向连接的传输: TCP



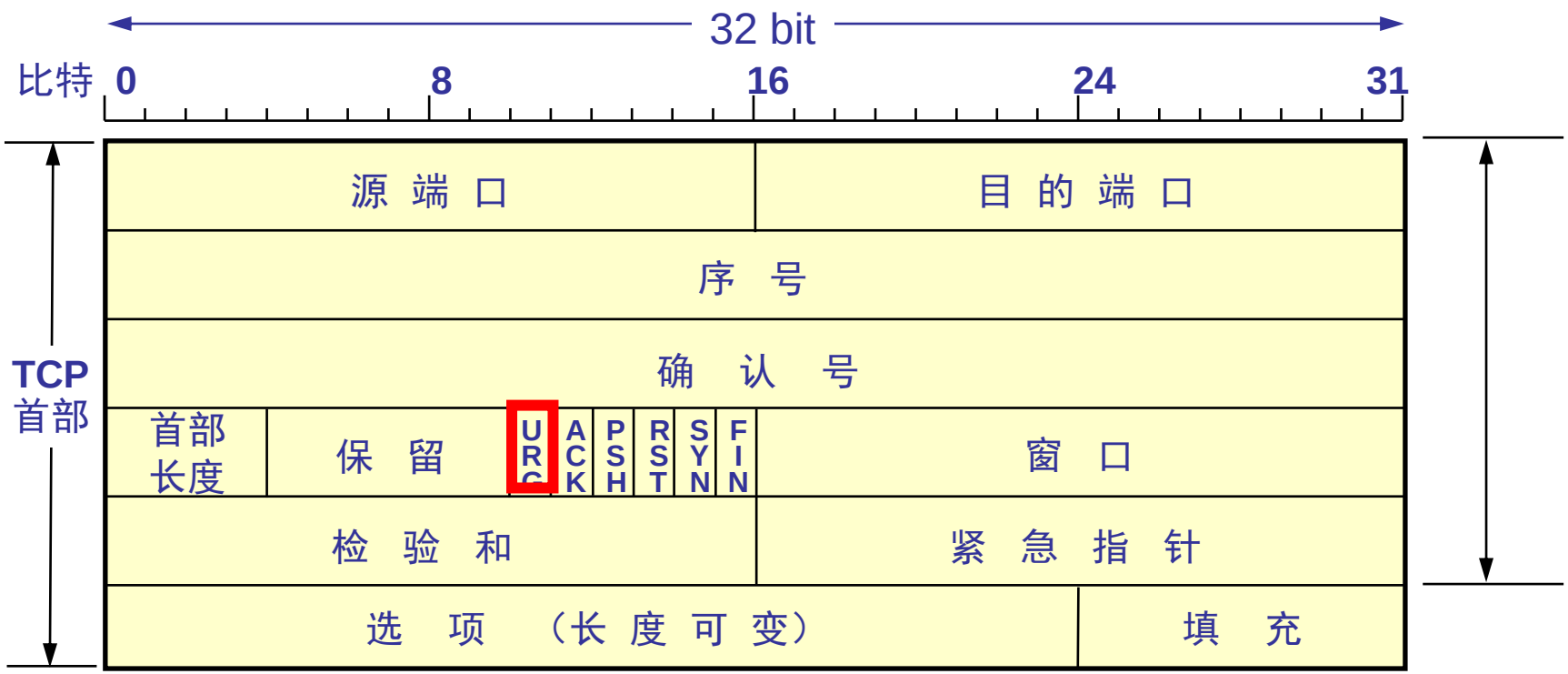
首部长度——占 4 bit，它指示意32bit的字为单位的**TCP首部长度**。
若选项字段为空，TCP首部典型长度为20字节。

3.5 面向连接的传输: TCP



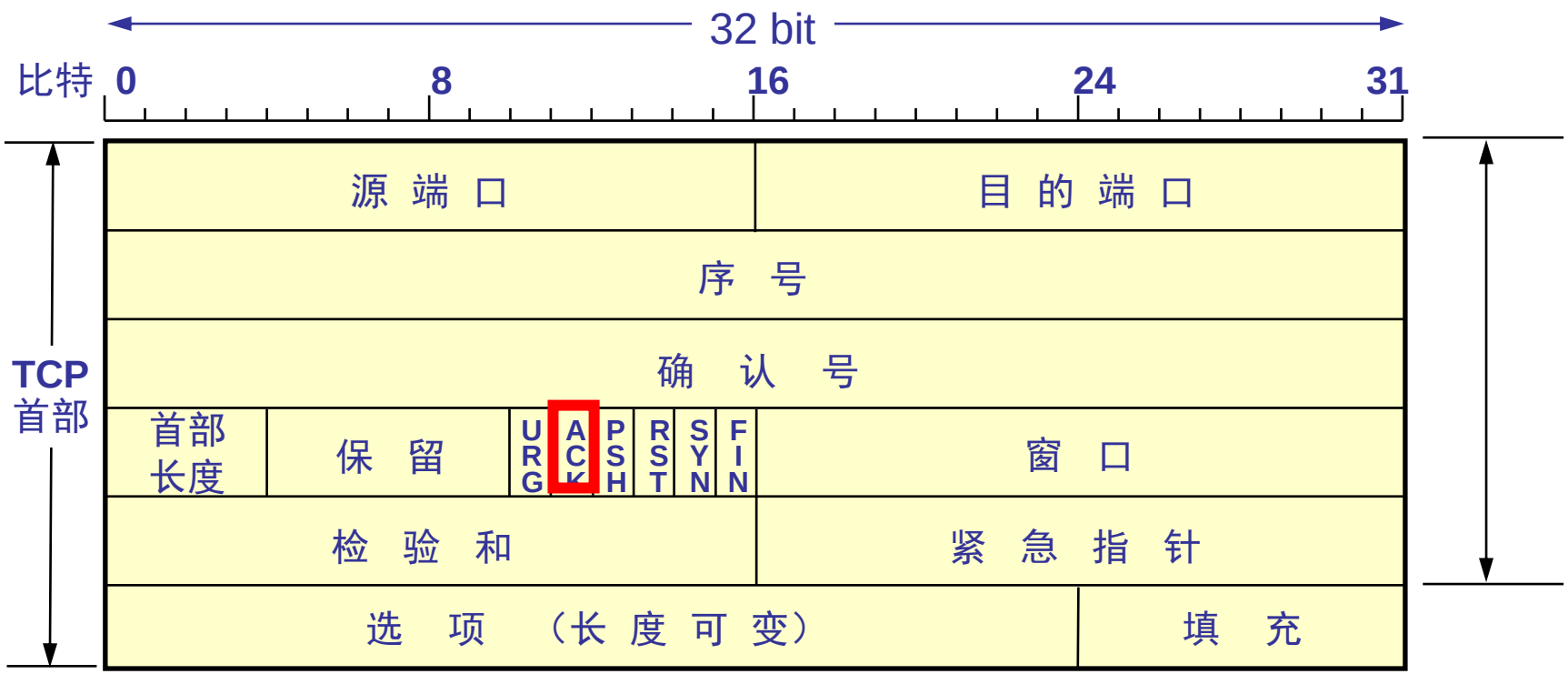
保留字段——占 6 bit，保留为今后使用，但目前应置为 0。

3.5 面向连接的传输: TCP



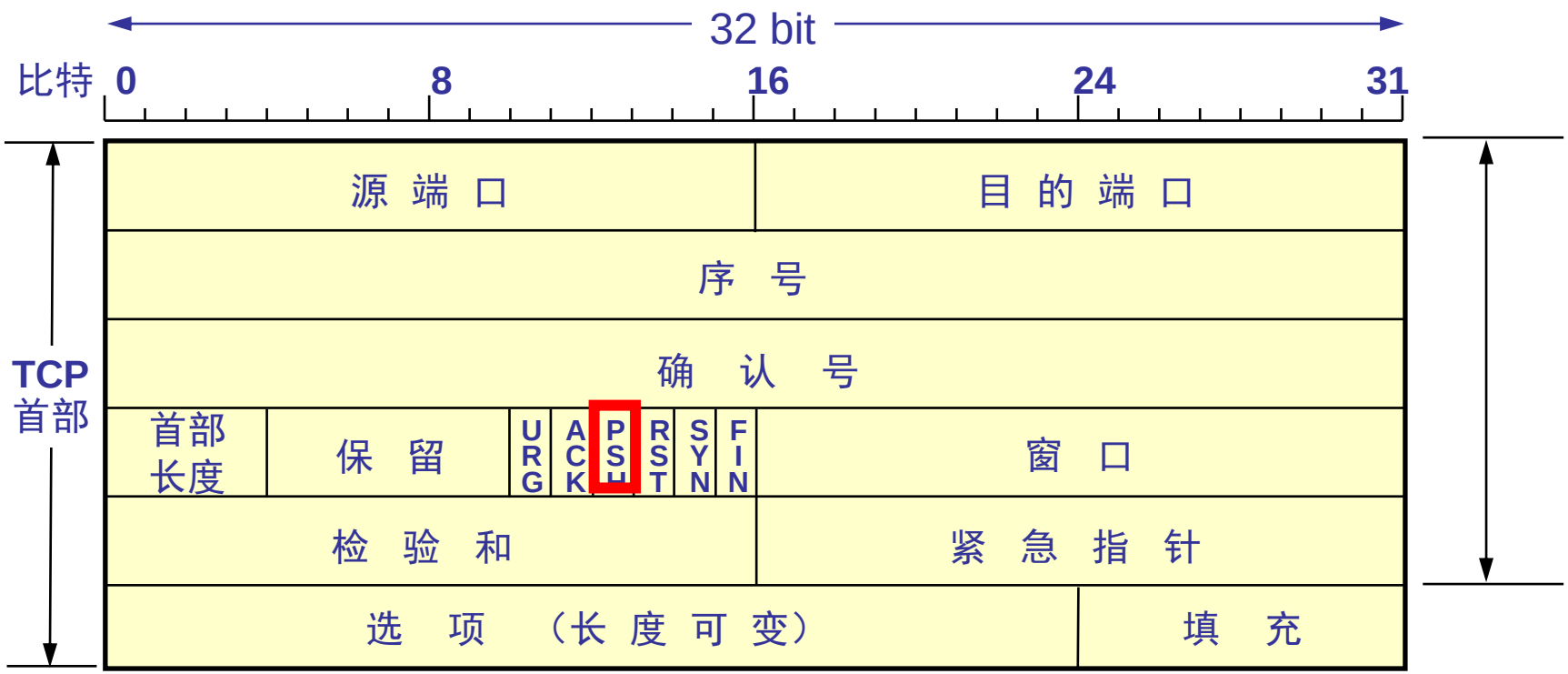
紧急比特 URG —— 当 URG = 1 时，表明紧急指针字段有效。它告诉系统此报文段中有紧急数据，应尽快传送。（一般不使用）

3.5 面向连接的传输: TCP



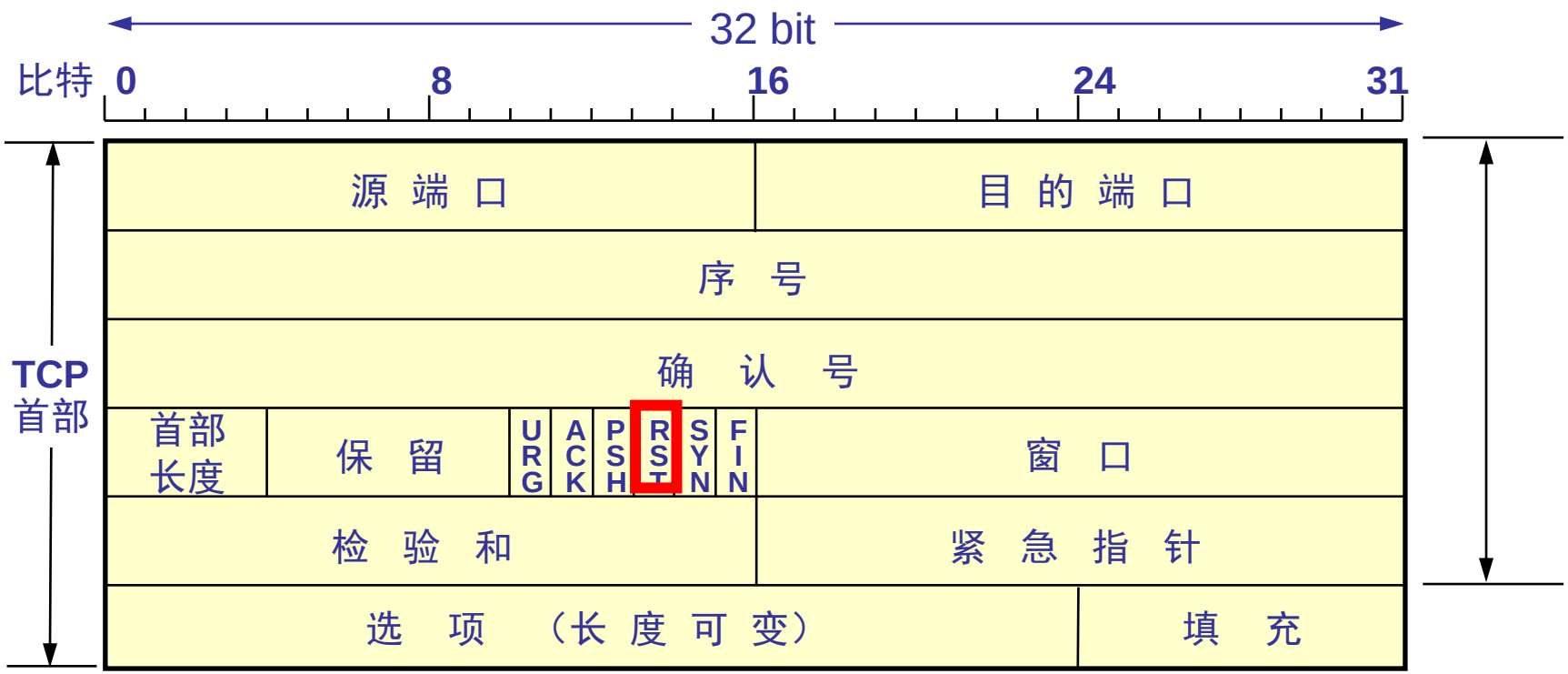
确认比特 ACK —— 只有当 ACK = 1 时确认号字段才有效。
当 ACK = 0 时，确认号无效。

3.5 面向连接的传输: TCP



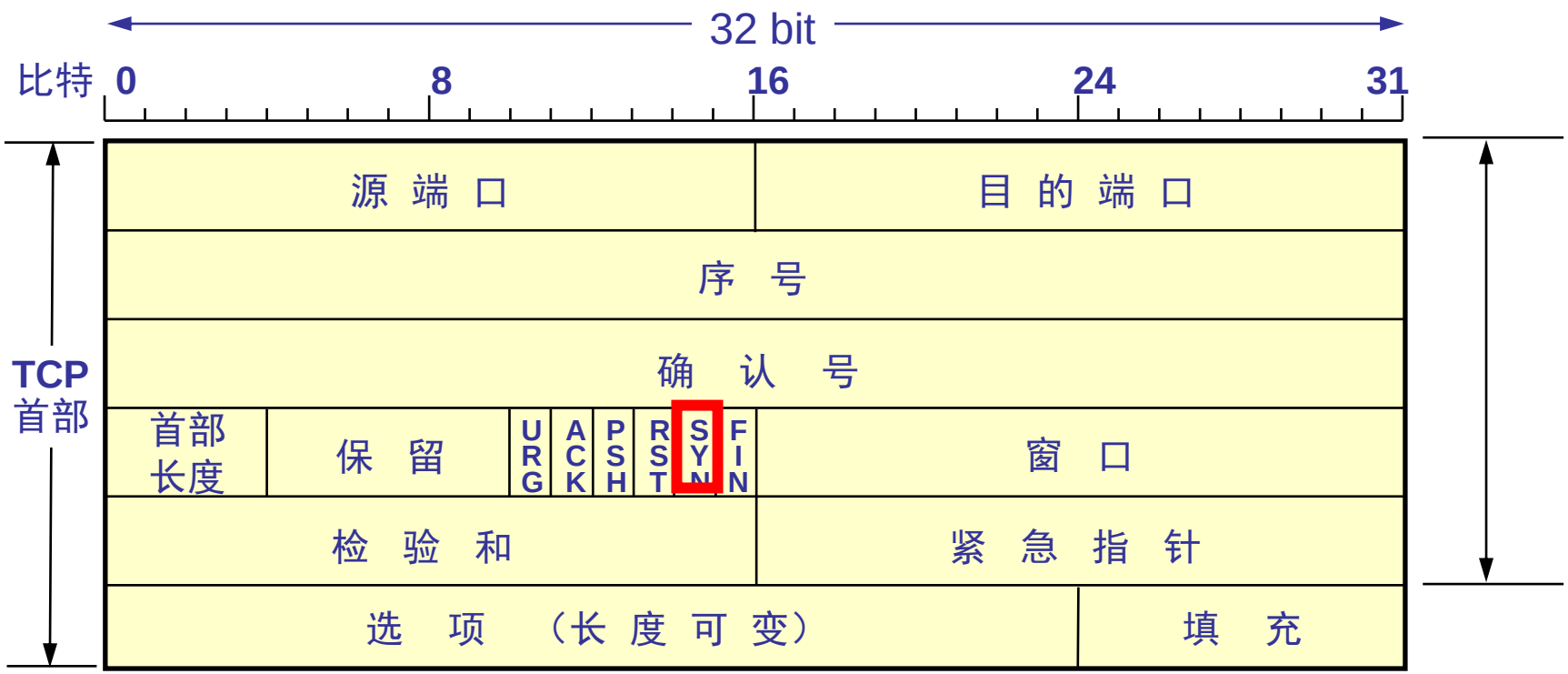
推送比特 PSH (PuSH) —— 接收 TCP 收到推送比特置 1 的报文段，就尽快地交付给接收应用进程，而不再等到整个缓存都填满了后再向上交付。

3.5 面向连接的传输: TCP



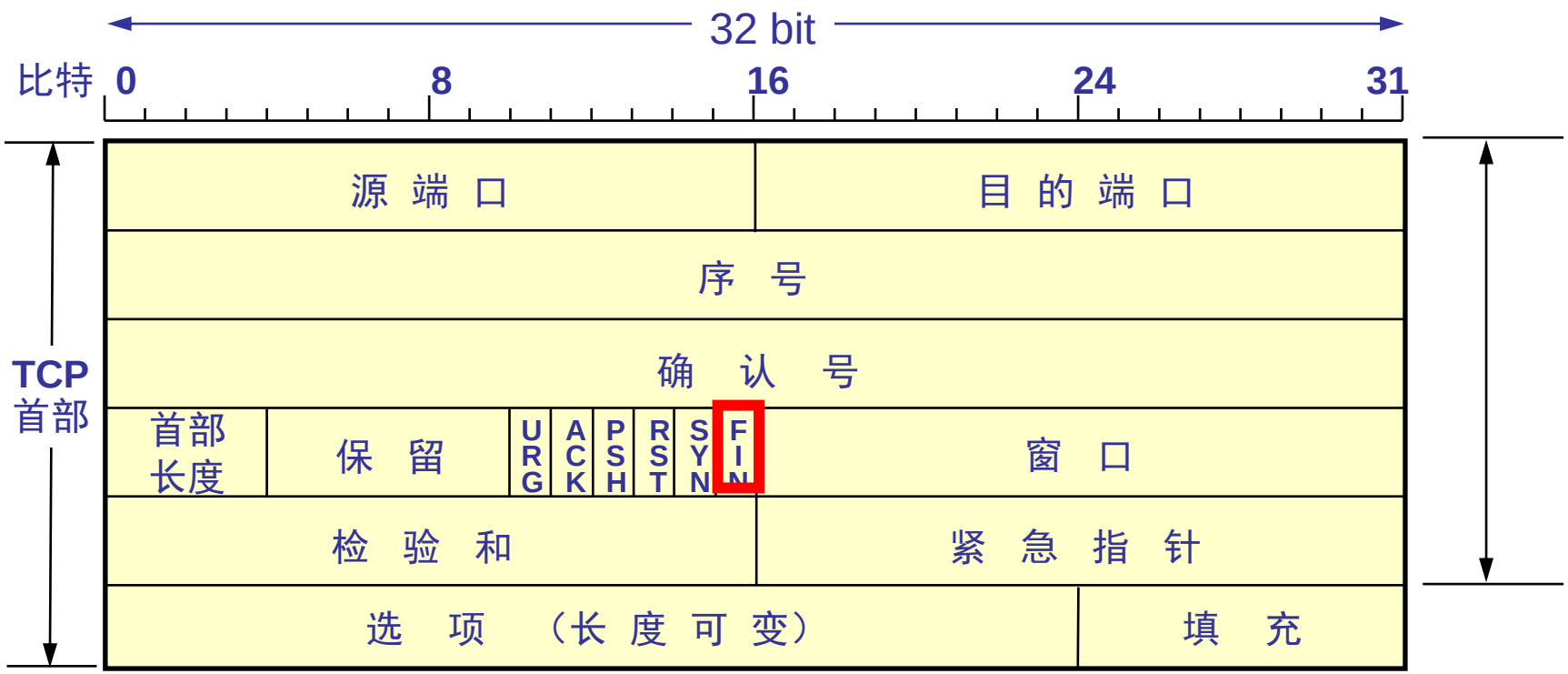
复位比特 RST (ReSeT) —— 当 RST = 1 时，表明 TCP 连接中出现严重差错（如由于主机崩溃或其他原因），必须释放连接，然后再重新建立运输连接。

3.5 面向连接的传输: TCP



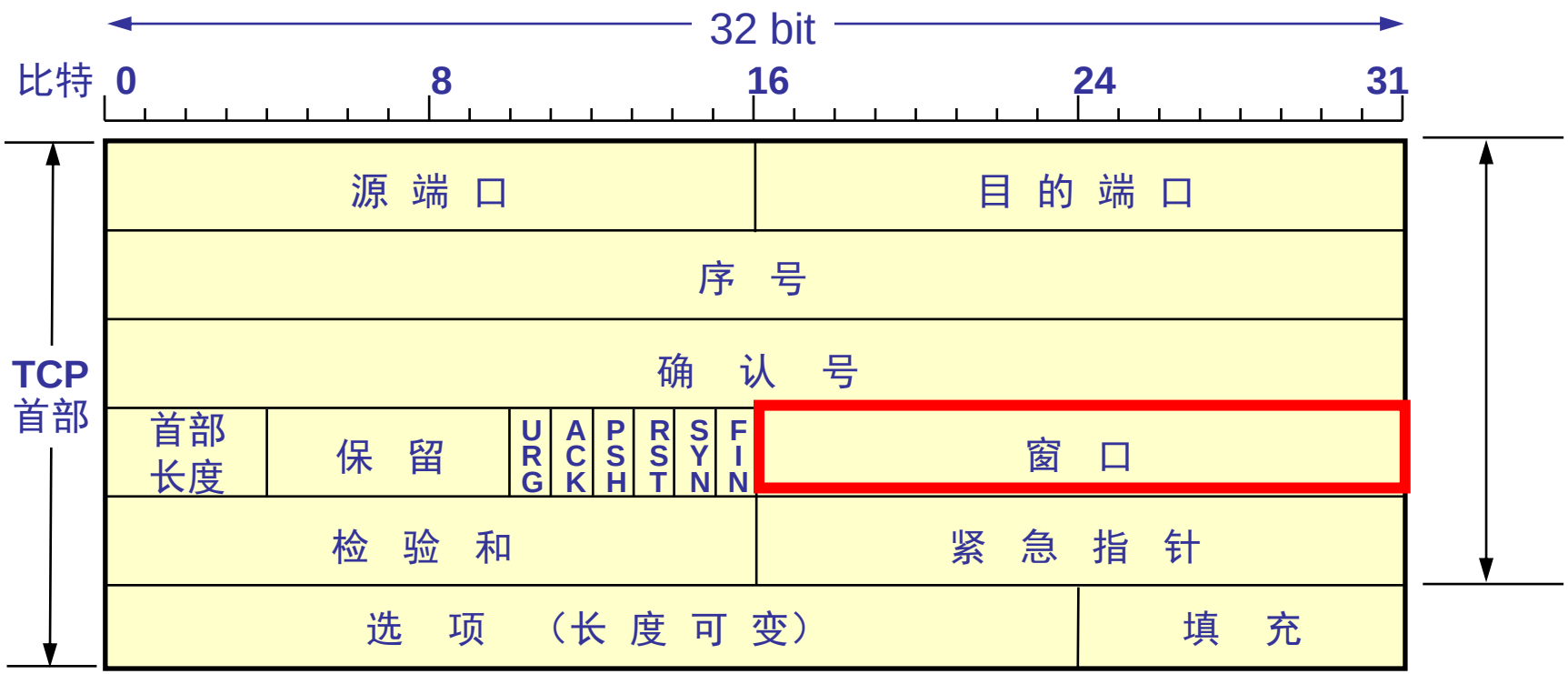
同步比特 SYN —— 同步比特 SYN 置为 1，就表示这是一个**连接请求**或**连接接受**报文。

3.5 面向连接的传输: TCP



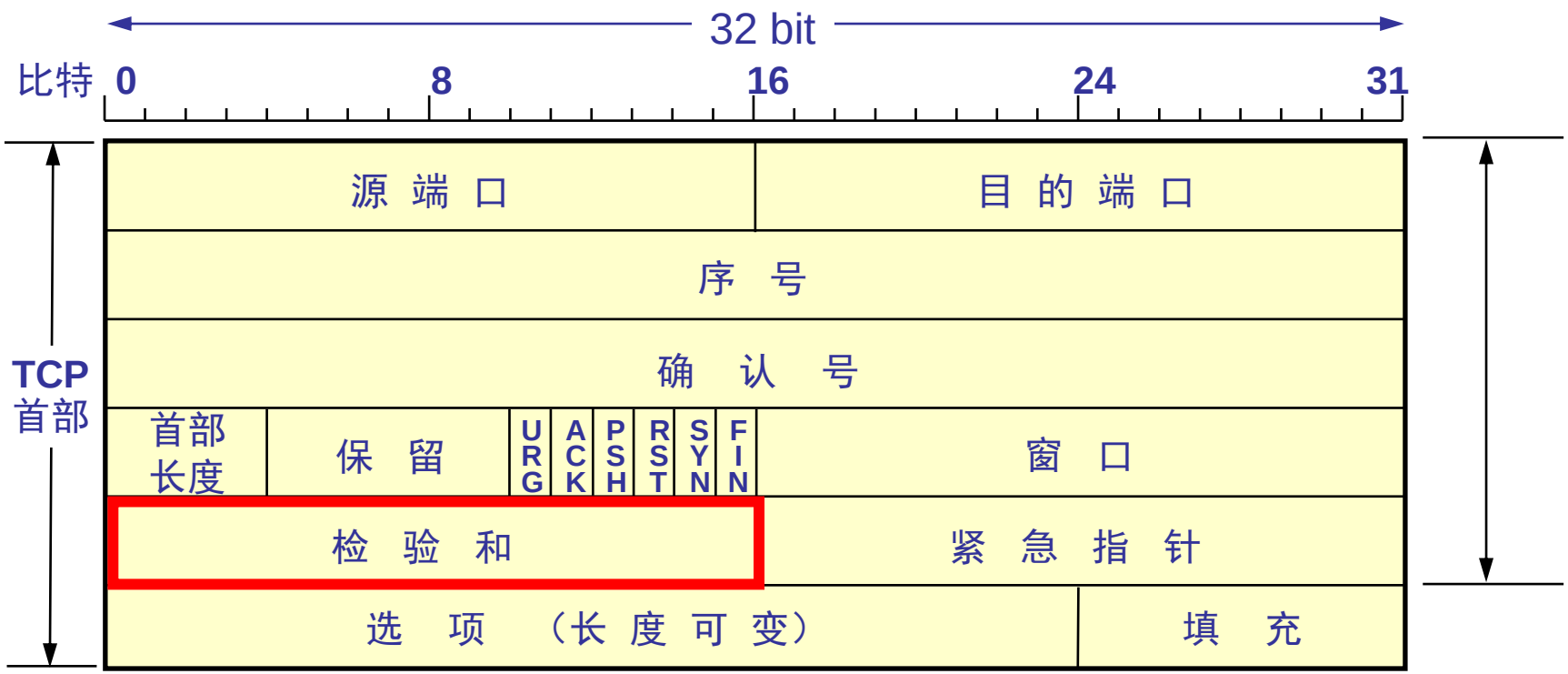
终止比特 FIN (FINal) —— 用来**释放一个连接**。当FIN = 1 时，表明此报文段的发送端的数据已发送完毕，并要求释放运输连接。

3.5 面向连接的传输: TCP



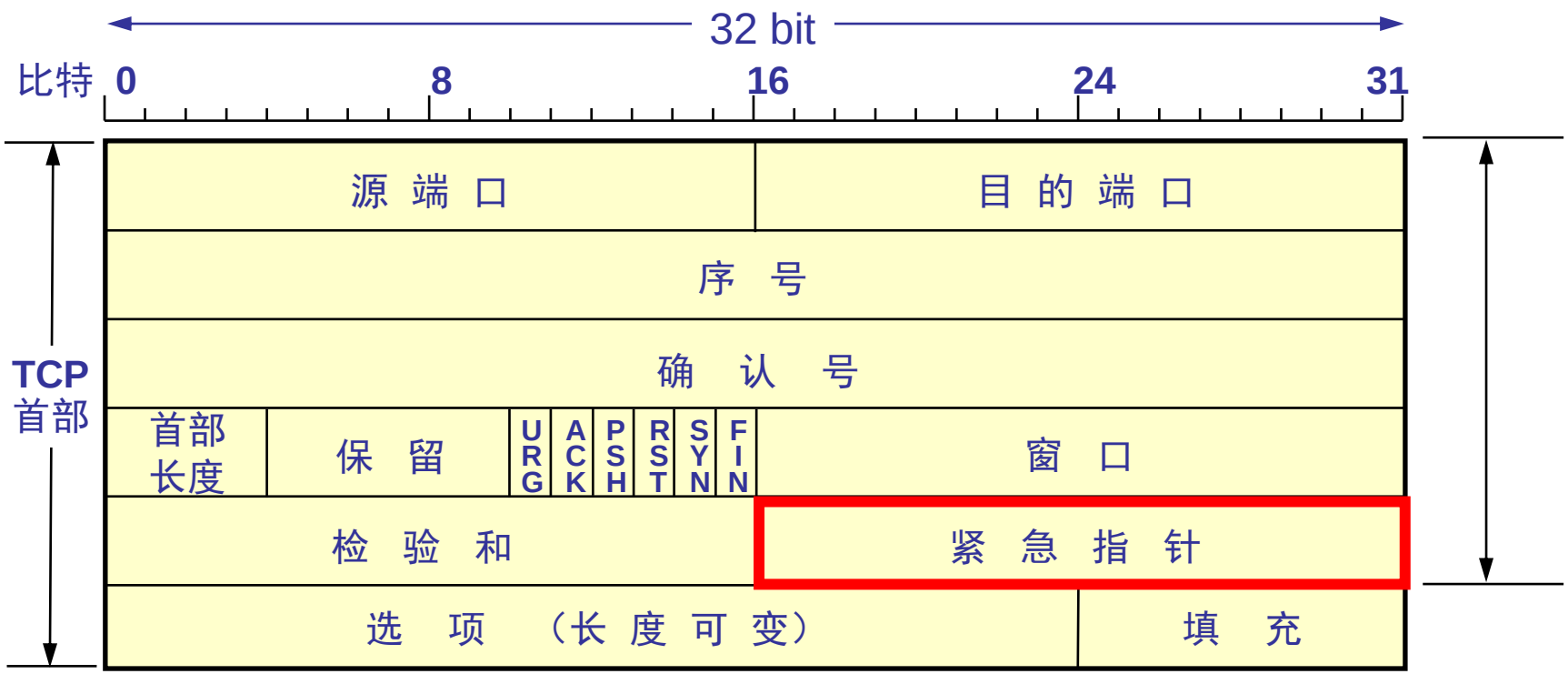
窗口字段 —— 占 2 字节。窗口字段用来控制对方发送的数据量，单位为字节。TCP 连接的一端根据设置的缓存空间大小确定自己的接收窗口大小，然后通知对方以确定对方的发送窗口的上限。

3.5 面向连接的传输: TCP



检验和 —— 占 2 字节。检验和字段检验的范围包括首部和数据这两部分。在计算检验和时，要在 TCP 报文段的前面加上 12 字节的伪首部。

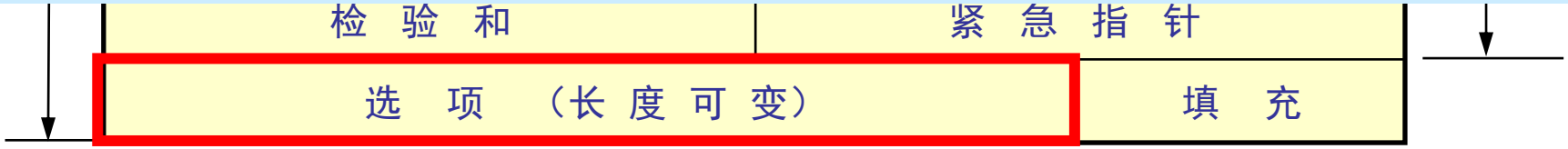
3.5 面向连接的传输: TCP



紧急指针字段 —— 占 16 bit。紧急指针指出在本报文段中的**紧急数据**的**最后一个字节的序号**。

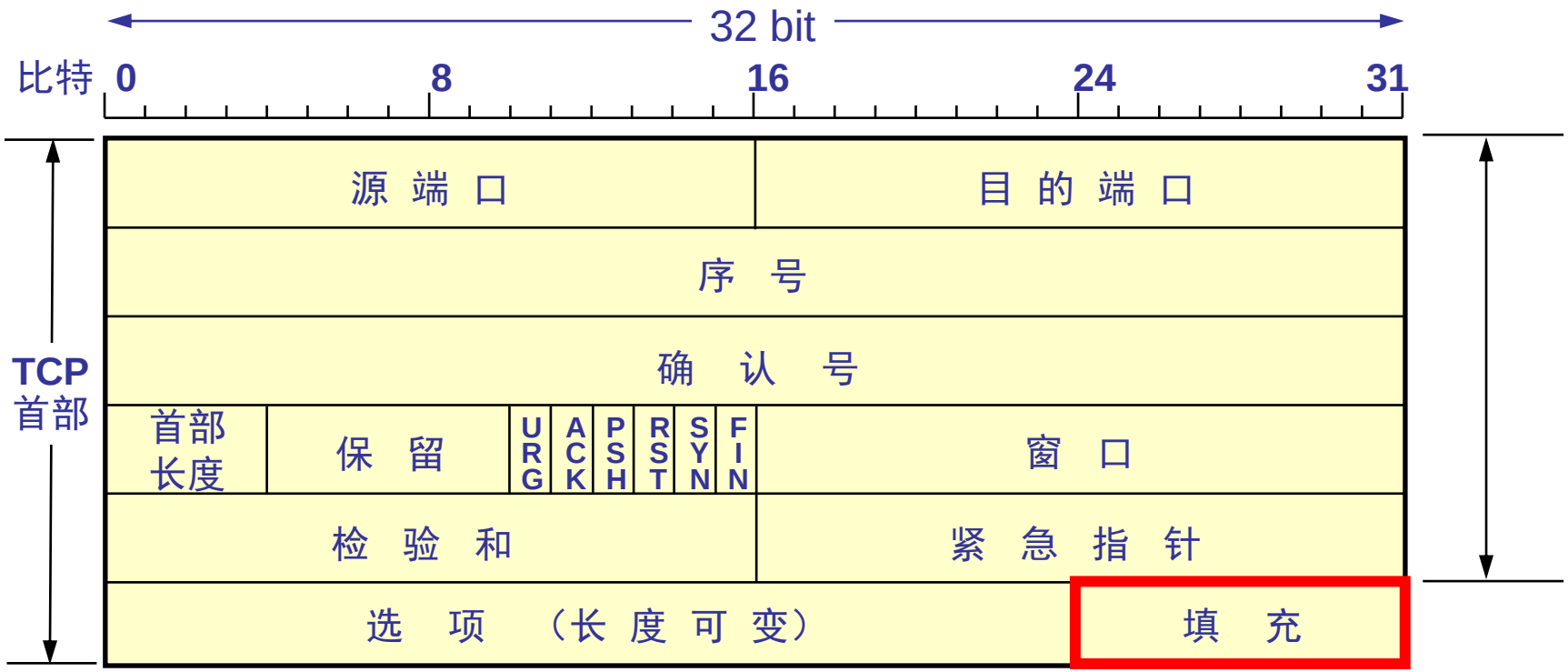
3.5 面向连接的传输: TCP

MSS 是 TCP 报文段中的数据字段的最大长度。
数据字段加上 TCP 首部
才等于整个的 TCP 报文段。



选项字段 —— 长度可变。TCP 只规定了一种选项，即最大报文段长度 MSS (Maximum Segment Size)。MSS 告诉对方 TCP：“我的缓存所能接收的报文段的数据字段的最大长度是 MSS 个字节。”

3.5 面向连接的传输: TCP



填充字段 —— 这是为了使整个首部长度是 4 字节的整数倍。

TCP超时的设置

- 如何设置TCP的超时?
  - 应该**大于RTT**
    - RTT：往返时延
    - 但 RTT是变化的
- 太短:
  - 造成不必要的重传
- 太长:
  - 对丢包反应太慢

◆如何估算 RTT

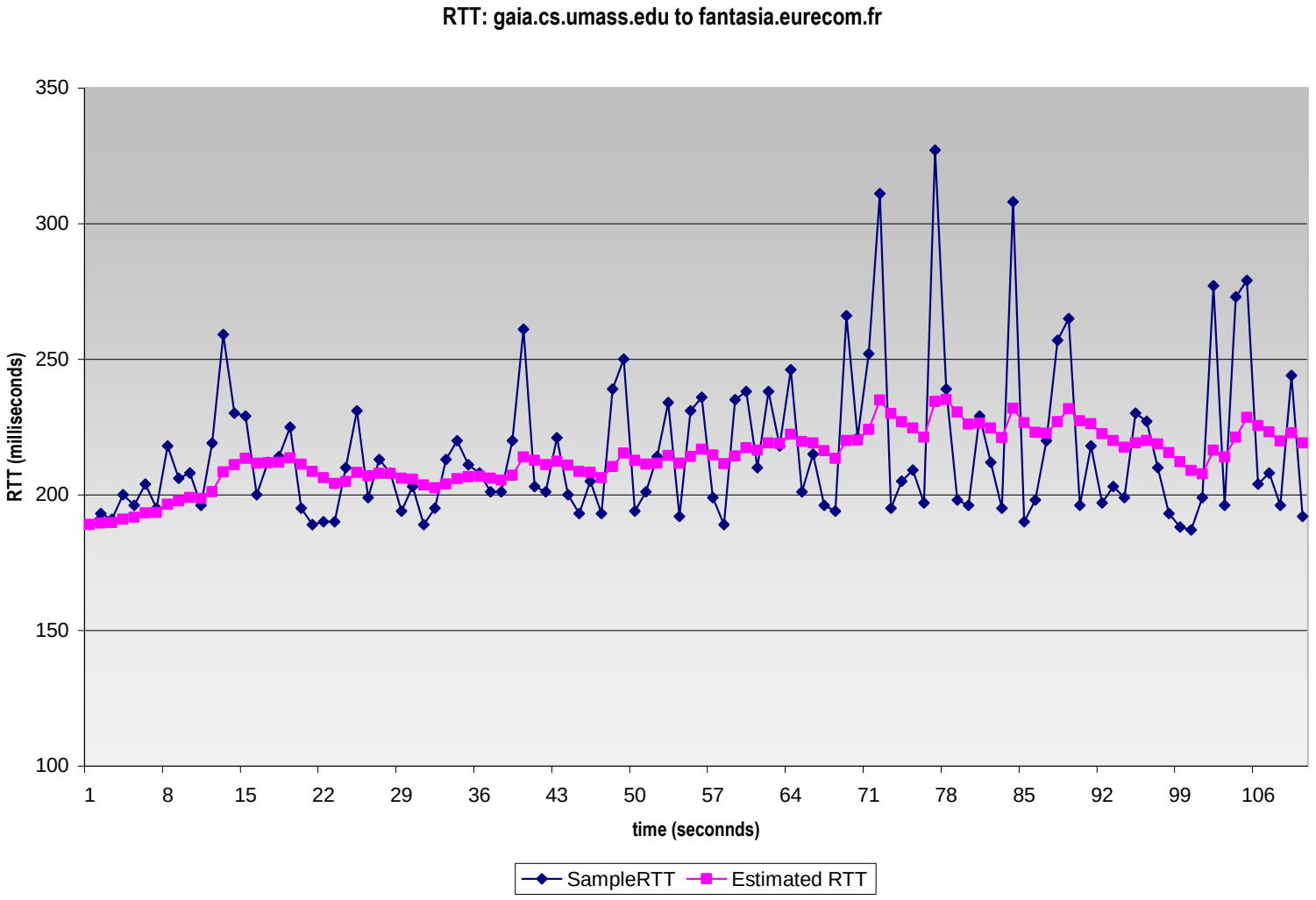
- ◆**样本RTT(SampleRTT)**：对报文段被发出到收到该报文段的确认之间的时间进行测量
  - ◆ 忽略重传
- ◆样本RTT会有波动，要使得估算RTT更平滑，需要将最近几次的测量进行平均，而非仅仅采用最近一次的SampleRTT

$$\text{EstimatedRTT} = (1-\alpha)*\text{EstimatedRTT} + \alpha*\text{SampleRTT}$$

参考值: $\alpha = 0.125$

3.5 面向连接的传输: TCP

◆RTT估计的一个例子



◆考虑RTT的波动，估计EstimatedRTT与SampleRTT的偏差

DevRTT = (1-β)*DevRTT +
β*|SampleRTT-EstimatedRTT|

(参考值, β = 0.25)
注意，第一次计算时，**DevRTT=0.5*SampleRTT**

TCP中的超时间隔为

TimeoutInterval = EstimatedRTT + 4*DevRTT

3.5 面向连接的传输：TCP


```
/* Called to compute a smoothed rtt estimate. The data fed to this
 * routine either comes from timestamps, or from segments that were
 * known _not_ to have been retransmitted [see Karn/Partridge
 * Proceedings SIGCOMM 87]. The algorithm is from the SIGCOMM 88
 * piece by Van Jacobson.
 * NOTE: the next three routines used to be one big routine.
 * To save cycles in the RFC 1323 implementation it was better to break
 * it up into three procedures. -- erics
 */
static void tcp_rtt_estimator(struct sock *sk, const __u32 mrtt)
{
        struct tcp_sock *tp = tcp_sk(sk);
        long m = mrtt; /* RTT */

        /*      The following amusing code comes from Jacobson's
         *      article in SIGCOMM '88.  Note that rtt and mdev
         *      are scaled versions of rtt and mean deviation.
         *      This is designed to be as fast as possible
         *      m stands for "measurement".
         *
         *      On a 1990 paper the rto value is changed to:
         *      RTO = rtt + 4 * mdev
         *
         * Funny. This algorithm seems to be very broken.
         * These formulae increase RTO, when it should be decreased, increase
         * too slowly, when it should be increased quickly, decrease too quickly
         * etc. I guess in BSD RTO takes ONE value, so that it is absolutely
         * does not matter how to _calculate_ it. Seems, it was trap
         * that VJ failed to avoid. 8)
         */
```

http://lxr.free-electrons.com/source/net/ipv4/tcp_input.c?v=2.6.32#L609

3.5 面向连接的传输: TCP

- 对RTT和RTO代码解读和分析（课后讨论）
- 参考
  - http://lxr.free-electrons.com/source/net/ipv4/tcp_input.c?v=2.6.32#L600

◆可靠的TCP数据传输

- ◆IP协议是不可靠的
- ◆TCP采用了3.4节阐述的数据可靠传输的方法
- ◆特别之处
  - ◆TCP编号采用按**字节**编号，而非按报文段编号
  - ◆TCP仅采用唯一（**单一**）的超时定时器
    - ◆假定每一个已发送但未被确认的报文段都与一个定时器相关联
    - ◆但定时器的管理需要相当大的开销

3.5 面向连接的传输: TCP


```
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
        start timer
    event: ACK received, 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 */
```

从应用程序接收数据

- 将数据封装入报文段中，每个报文段都包含一个序号
- 序号是该报文段第一个数据字节的字节流编号
- 启动定时器
- 超时间隔: TimeOutInterval

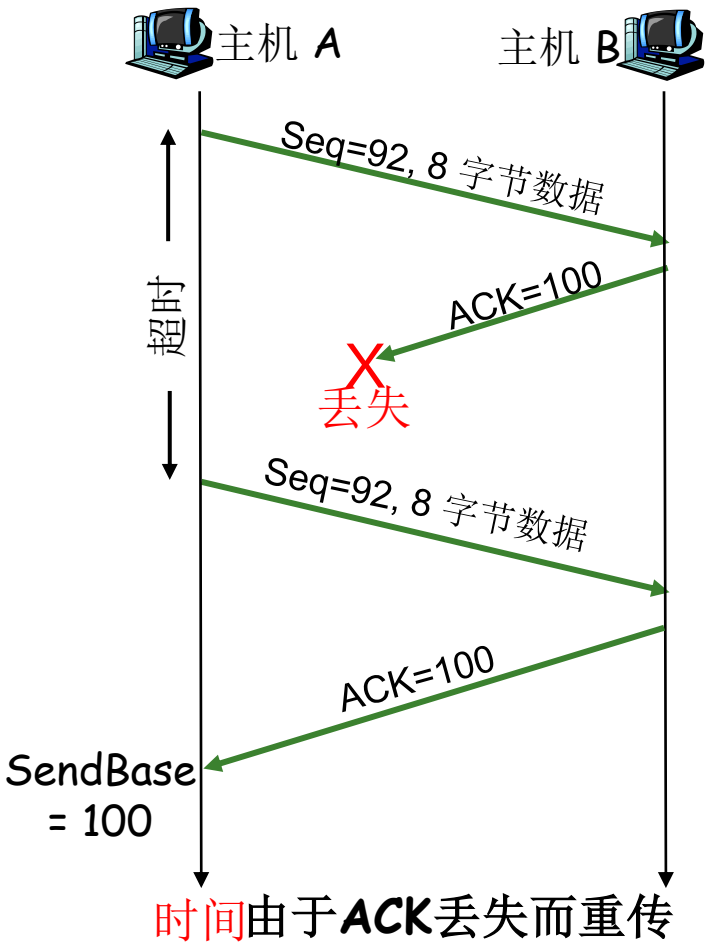
超时

- 重传认为超时的报文段
- 重启定时器

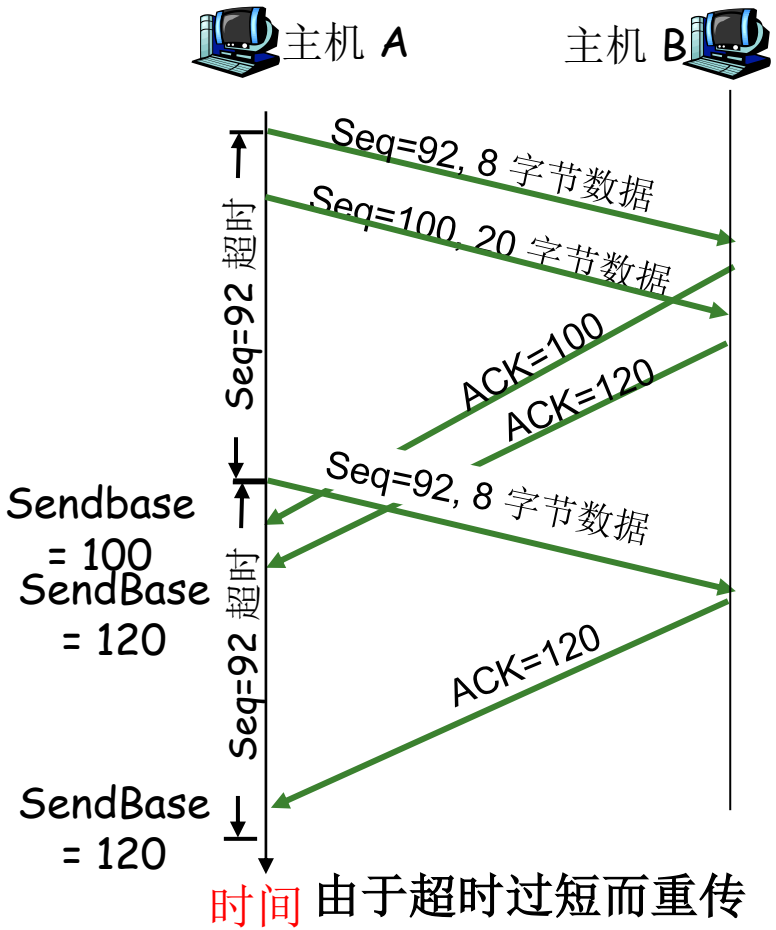
收到ACK

- 如果是对以前的未确认报文段的确认
  - 更新SendBase
  - 如果当前有未被确认的报文段，TCP还要重启定时器

◆TCP的几种重传情况

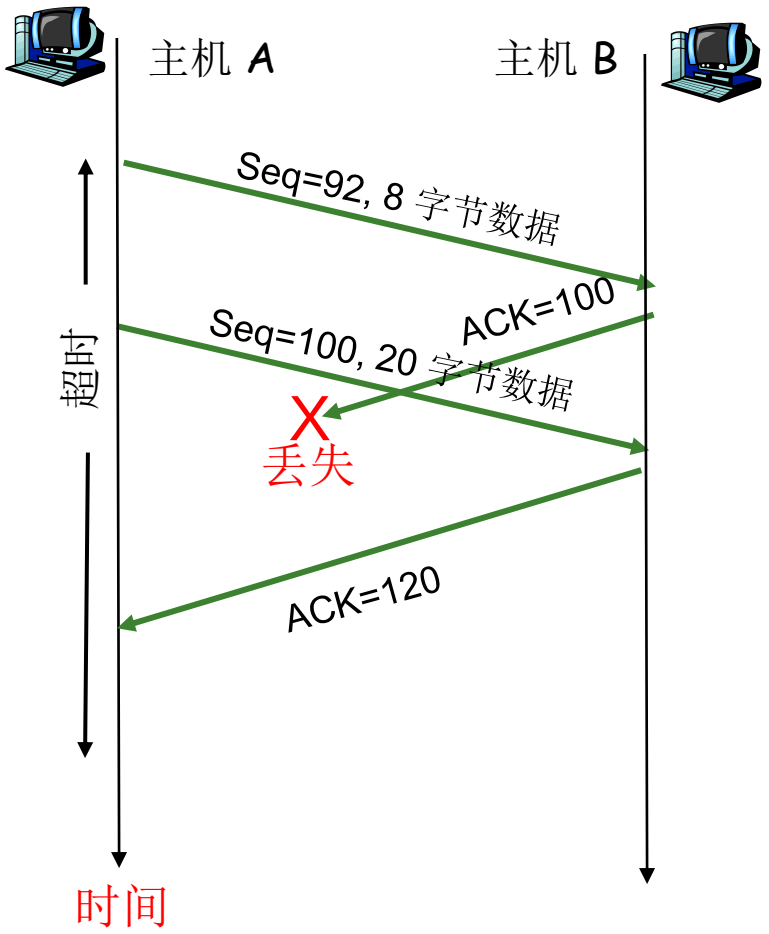


由于ACK丢失而重传



由于超时过短而重传

3.5 面向连接的传输：TCP



Seq=100先到了怎么办？

Seq=92 到了怎么办？

累积确认避免了第一个报文的重传

◆产生TCP ACK的建议（RFC1122、2581）

接收方事件	TCP接收方 动作
所期望序号的报文段按序到达。所有在期望序号及其以前的数据都已经被确认	延迟的ACK。对另一个按序报文段的到达最多等待500ms，如果下一个按序报文段在这个时间间隔内没有到达，则发送一个ACK
所期望序号的报文段按序到达。另一个按序报文段等待发送ACK	立即发送单个累积ACK，以确认两个按序报文段
比期望序号大的失序报文段到达，检测出数据流中的间隔	立即发送冗余ACK，指明下一个期待字节的序号(也就是间隔的低端字节序号)
能部分或完全填充接收数据间隔的报文段到达	倘若该报文段起始于间隔的低端，则立即发送ACK

◆快速重传

- ◆超时周期往往太长
  - ◆ 增加重发丢失分组的延时
- ◆通过**重复的ACK**检测丢失报文段
  - ◆ 发送方常要连续发送大量报文段
  - ◆ 如果一个报文段丢失，会引起很多连续的重复ACK.
- ◆如果发送收到一个数据的**3个ACK**，它会认为确认数据之后的报文段丢失
  - ◆ **快速重传**: 在超时到来之前重传报文段

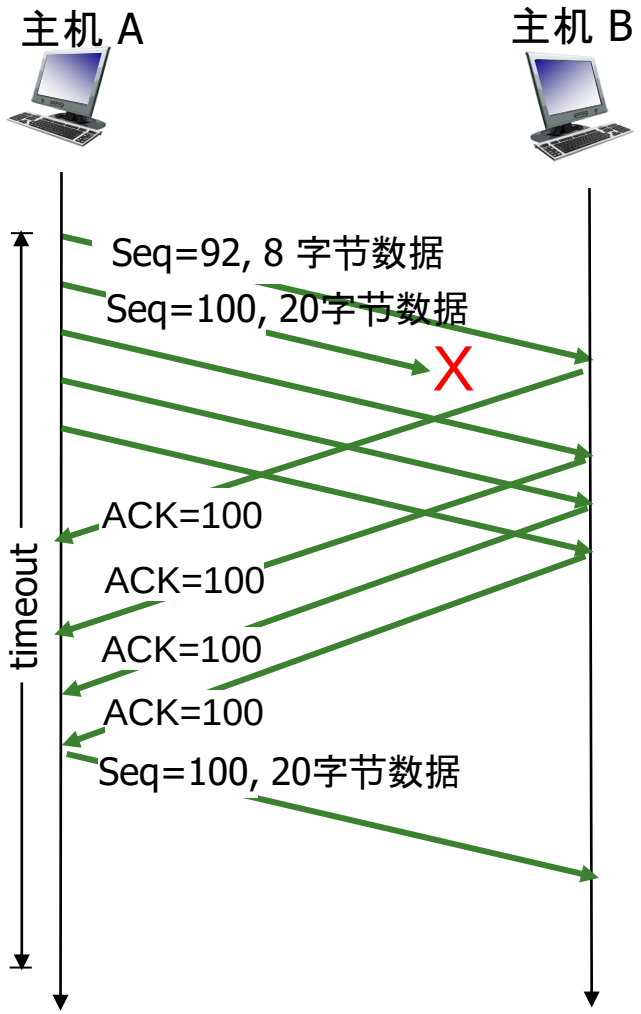
◆快速重传的算法

```
event: ACK received, with ACK field value of y
            if (y > SendBase) {
                SendBase = y
                if (there are currently not-yet-acknowledged segments)
                    start timer
            }
            else {
                increment count of dup ACKs received for y
                if (count of dup ACKs received for y = 3) {
                    resend segment with sequence number y
                }
```

重复的ACK报文

快速重传

◆快速重传



快速重传和超时重传分别代表了什么样的网络状况?

快速重传：收到3个重复的ACK立即重传

- 超时间隔加倍
  - 每一次TCP重传均将下一次超时间隔设为先前值的两倍
    - **Why?**
    - **若持续重传分组，加剧拥塞**
    - **超时时间呈指数型增长**
    - **形式受限的拥塞控制**
  - 超时间隔由EstimatedRTT和DevRTT决定
    - 发送以下两个事件，超时间隔重新计算
      - 收到上层应用的数据
      - 收到对未确认数据的ACK

TCP流量控制

背景

TCP接收方有一个缓存，所有上交的数据全部缓存在里面

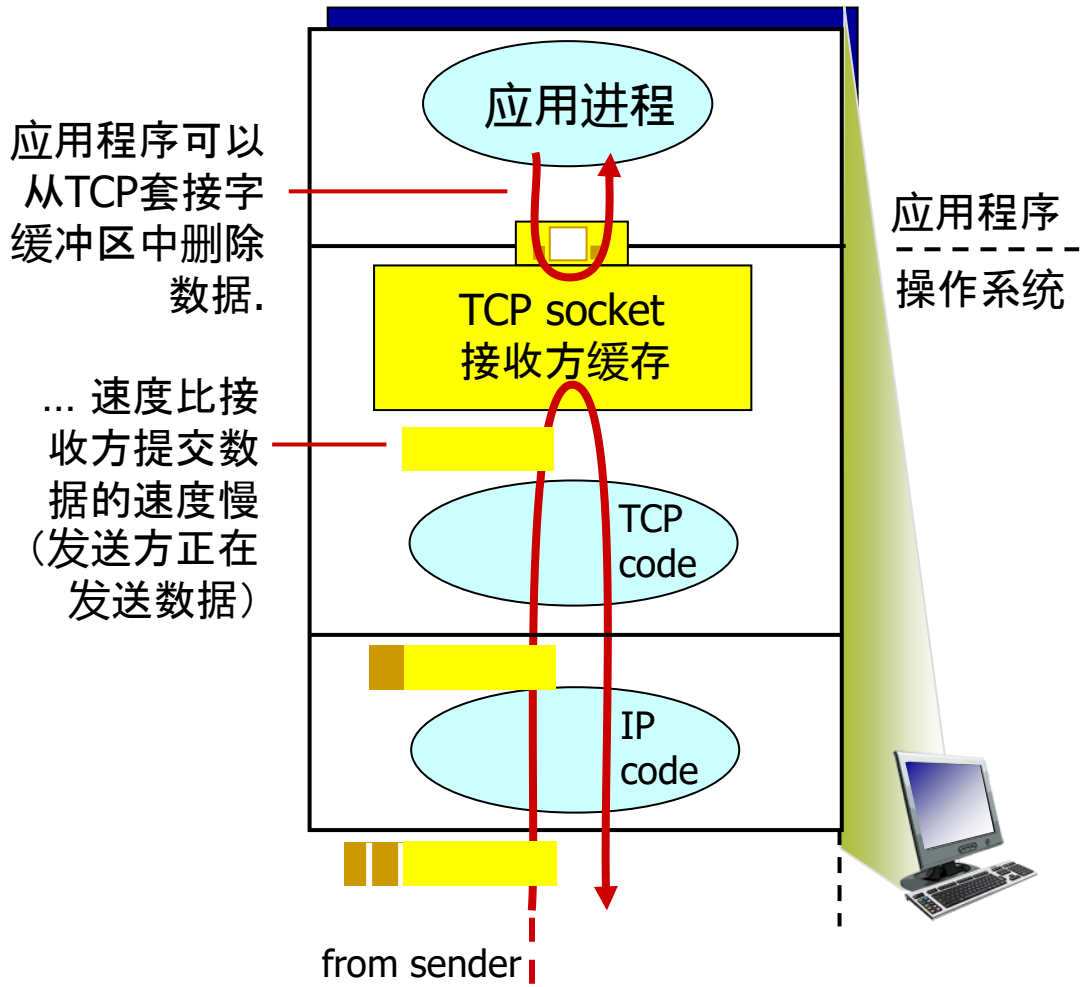
应用进程从缓冲区中读取数据可能很慢

目标

发送方不会由于传得太多太快而使得接收方缓存溢出

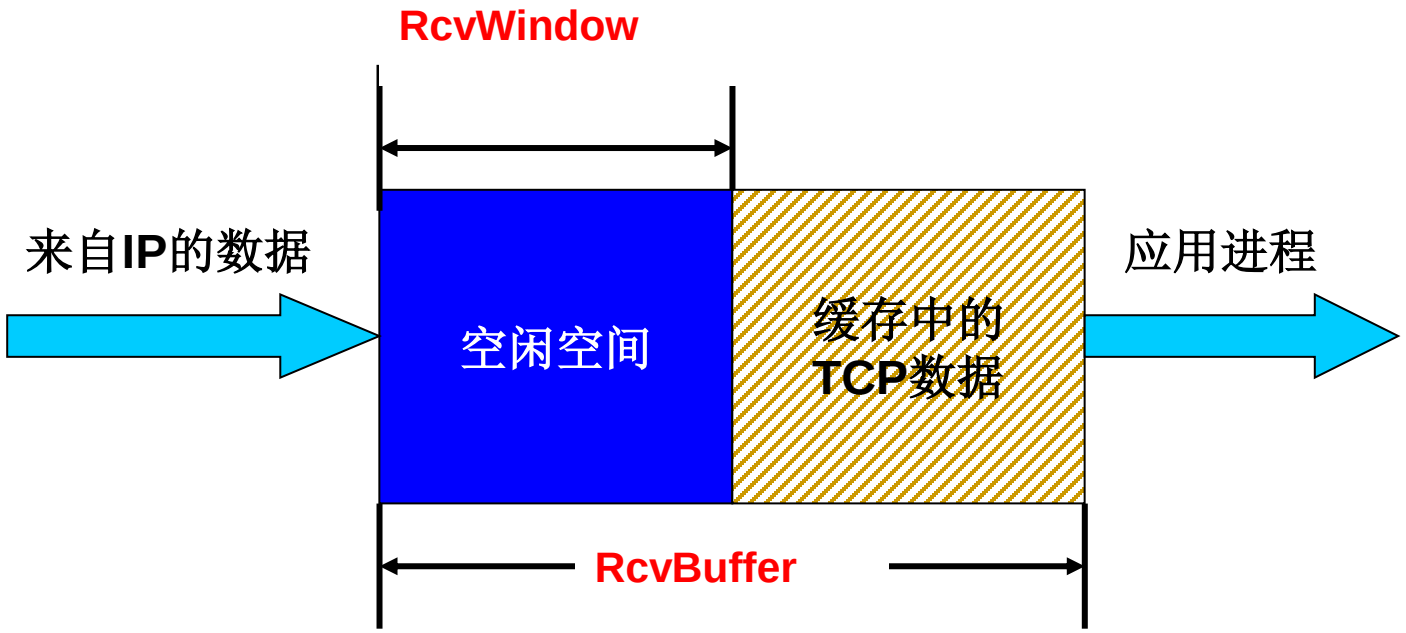
手段

接收方在反馈时，将缓冲区剩余空间的大小填充在报文段首部的窗口字段中，通知发送方



接收方协议堆栈

◆窗口值的计算



接收方： LastByteRcvd – LastByteRead ≤ RcvBuffer

RcvWindows = RcvBuffer – [LastByteRcvd - LastByteRead]

发送方： LastByteSent – LastByteAcked ≤ RcvWindow

◆思考一个问题

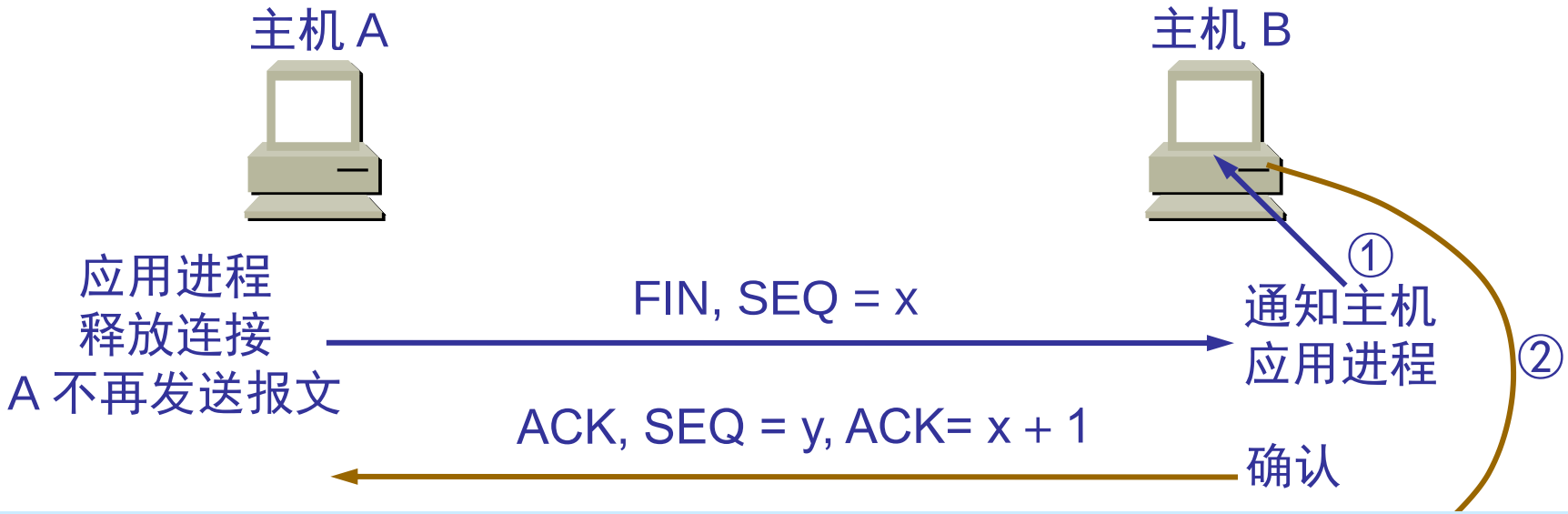
- ◆接收窗口的大小是随数据（应答）通知发送方，若接收方缓存满，且接收方不发送数据给发送方时，则发送方不再收到任何来自接收方的数据。而当接收方缓存发生变化（清空），会发生什么现象？
  - ◆ 接收方通知发送方RcvWindow为0，且接收方无任何数据传送给发送方
- ◆解决方案
  - ◆ 发送方持续向接受方发送只有**一个字节数据**的报文段，目的是试探

◆TCP连接的建立


主机 A
主机 B
主动打开
被动打开
连接请求
SYN, SEQ = x
SYN, ACK, SEQ = y, ACK= x + 1
确认
确认
ACK, SEQ = x + 1, ACK = y + 1

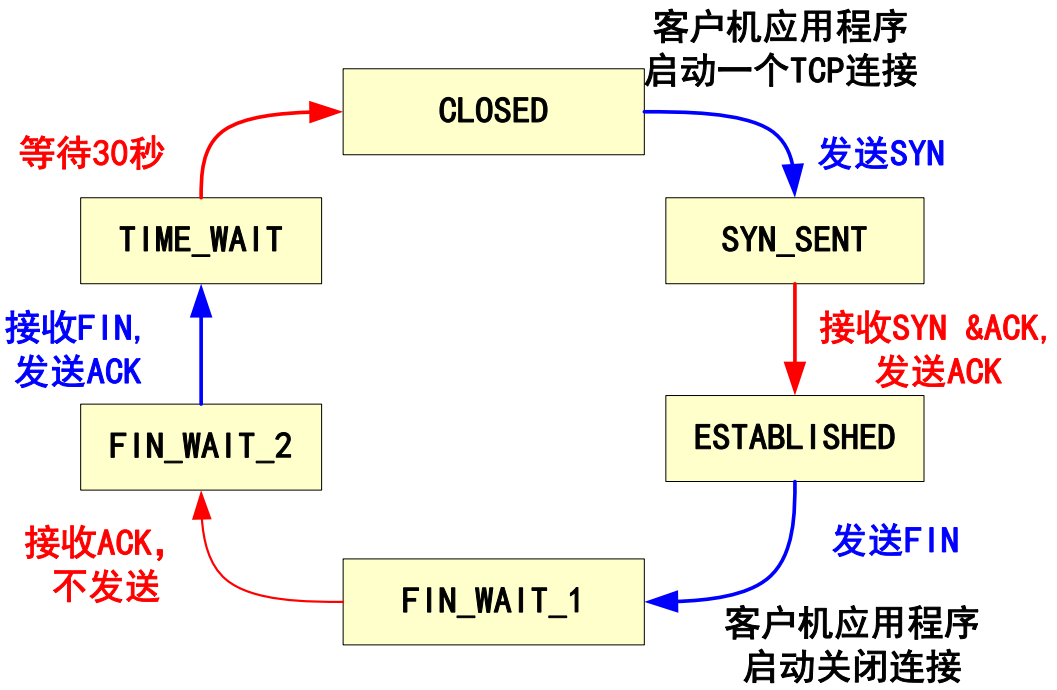
Wireshark
演示

至此，整个连接已经全部释放。

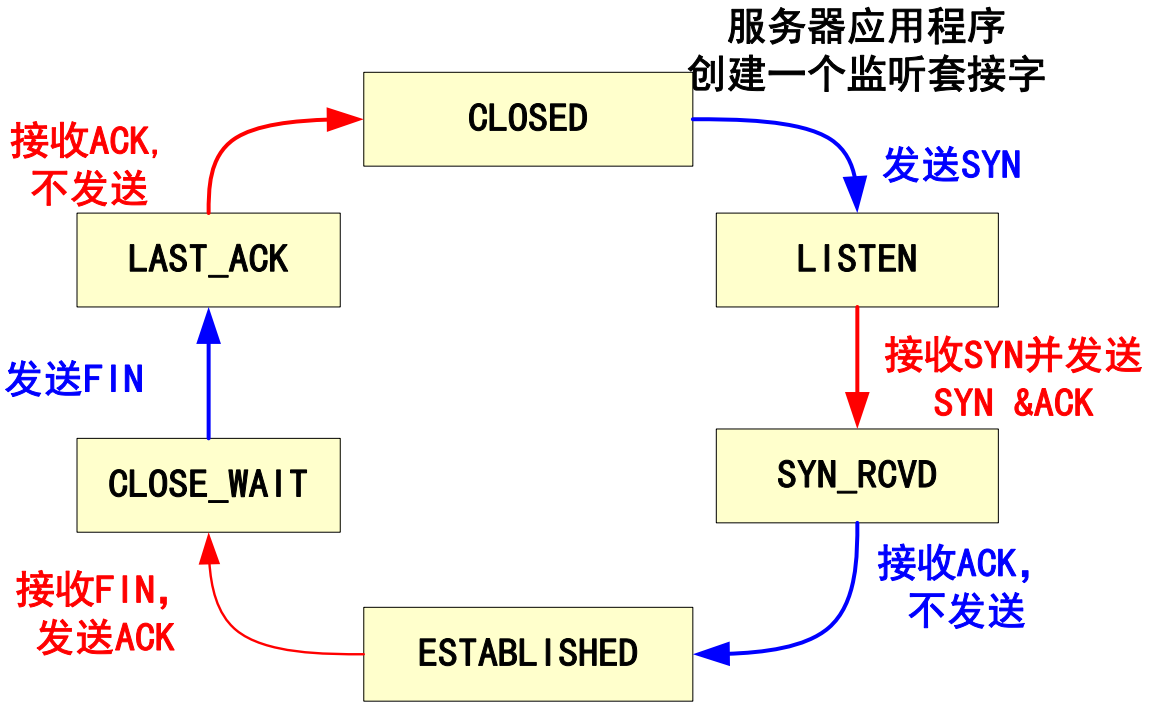


从 A 到 B 的连接就释放了，连接处于半关闭状态。
相当于 A 向 B 说：
“我已经没有数据要发送了。
但你如果还发送数据，我仍接收。”

◆TCP连接管理的状态序列



客户机TCP状态序列



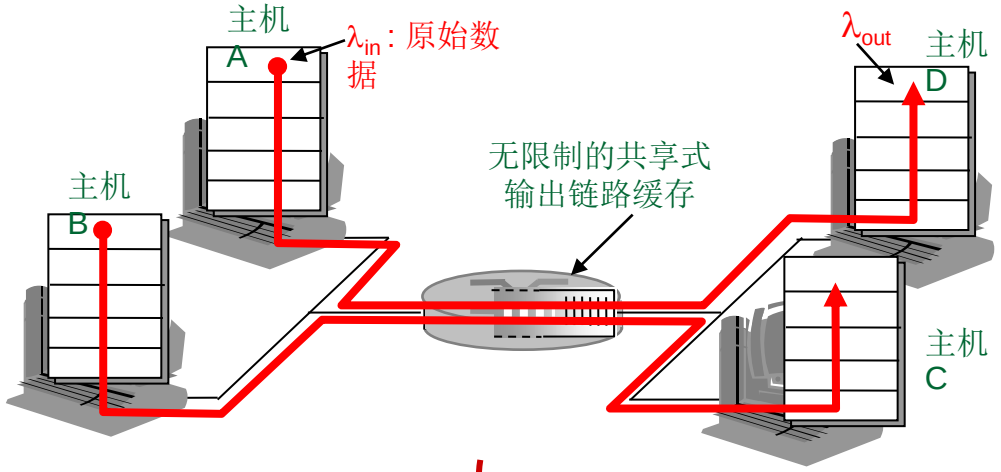
服务器TCP状态序列

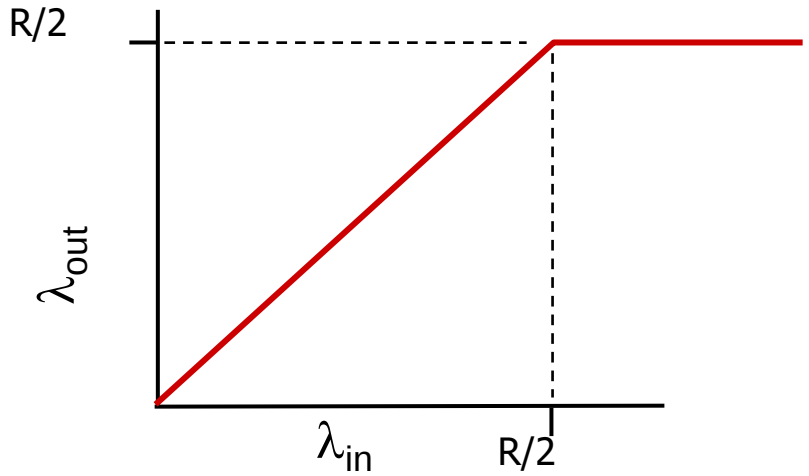
◆拥塞的基本知识

- ◆非正式定义: “过多的源发送了过多的数据，超出了网络的处理能力”
- ◆不同于流量控制!
- ◆现象:
  - ◆丢包 (路由器缓冲区溢出)
  - ◆延时长 (在路由器缓冲区排队)

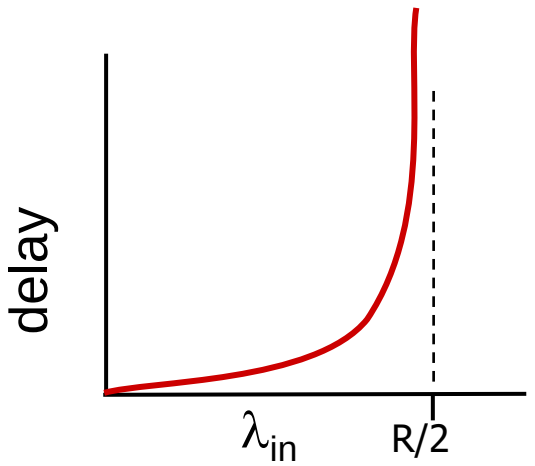
情境1

- 两个发送方，两个接受方
- 一个具有无限大缓存的路由器
- 没有重传
- 链路容量为R





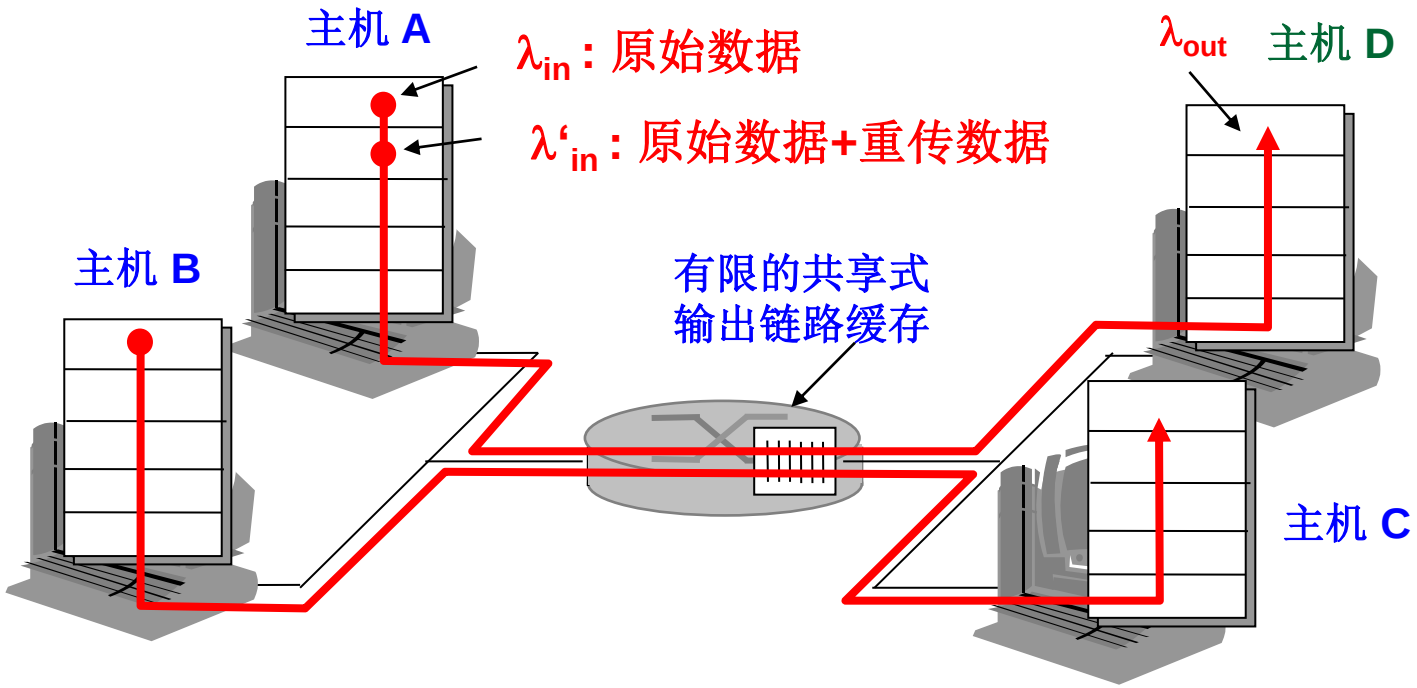
- 每连接的最大吞吐量: R/2



- 当分组到达速率接近链路容量时，分组经历的巨大排队时延
- **拥塞代价**

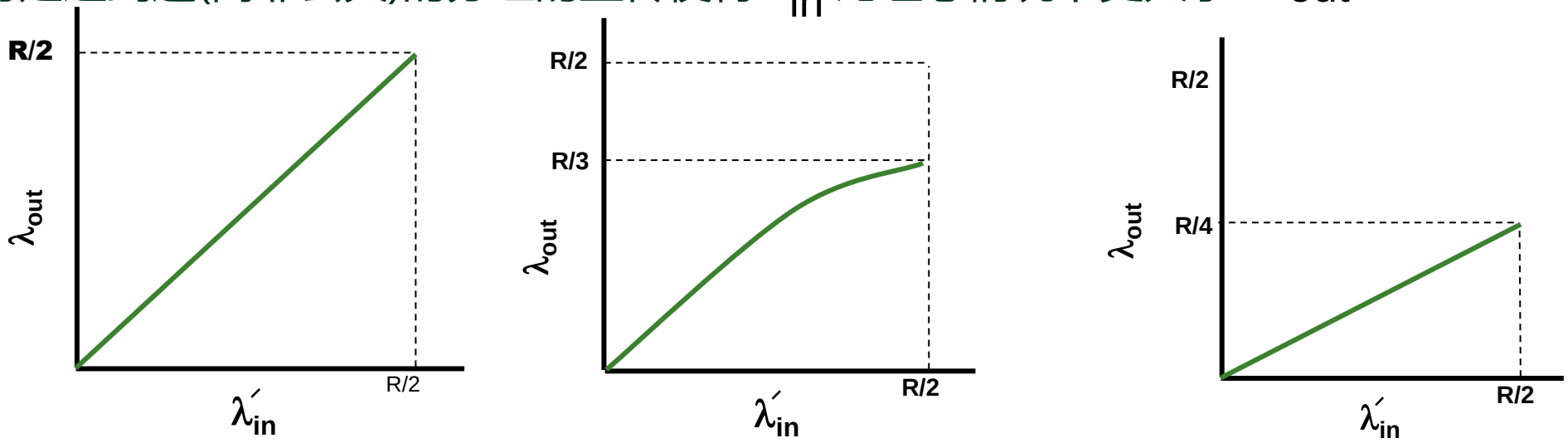
◆情境2

- ◆一个具有**有限** 缓存的路由器
- ◆发送方对丢失的分组进行重传



3.6 拥塞控制原理

- 设计期望: $\lambda_{in} = \lambda_{out}$ (goodput)
- “理想” 的重传是仅仅在丢包时才发生重传: $\lambda'_{in} > \lambda_{out}$
- 对延迟到达(而非丢失)的分组的重传使得 λ'_{in} 比理想情况下更大于 λ_{out}



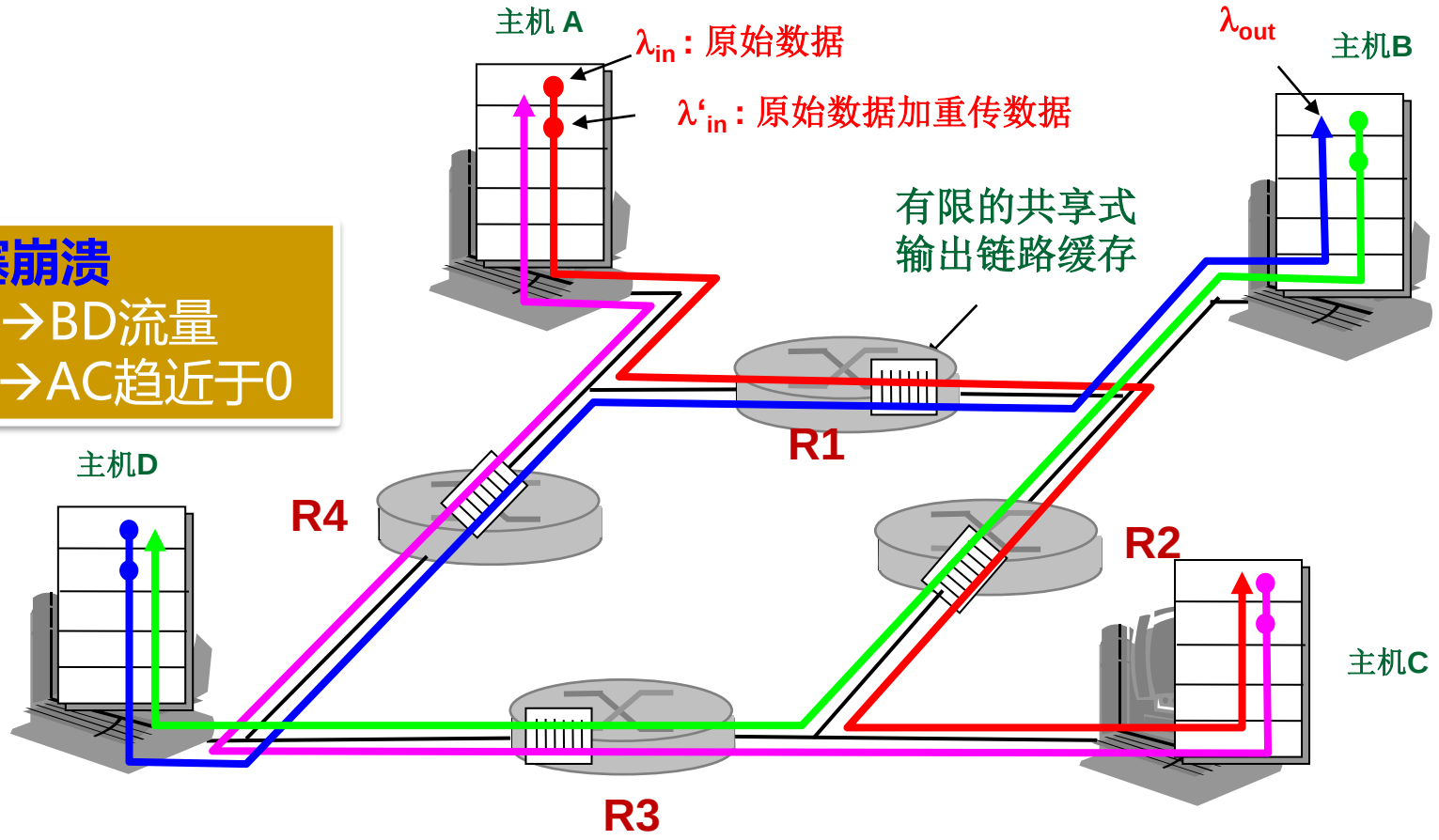
拥塞的“开销”：

- 发送方必须**重传**以**补偿**因为缓存溢出而丢失的分组
- 发送方在遇到大时延时所进行的**不必要重传**会引起路由器转发不必要的分组拷贝而**占用其链路带宽**

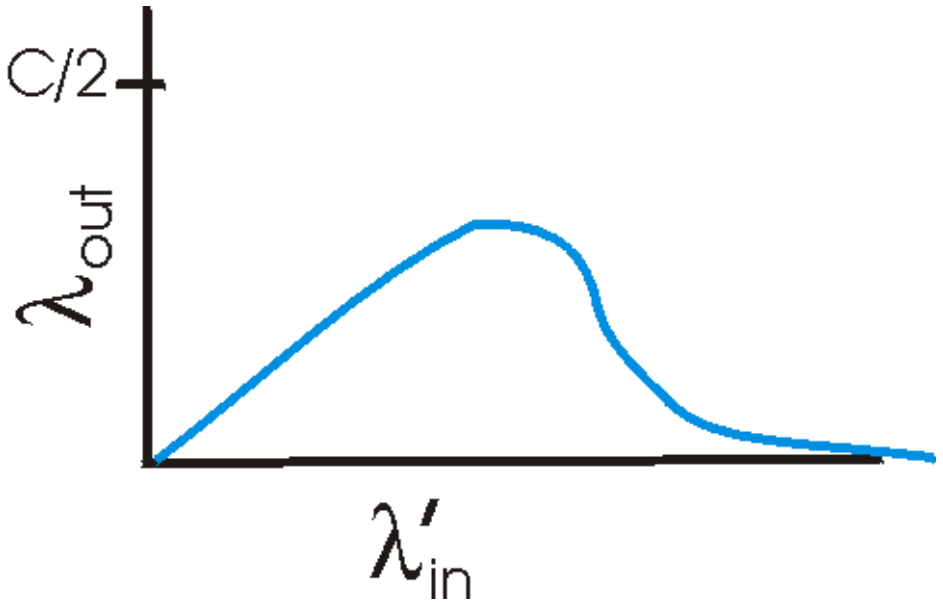
◆情境3

- ◆四个发送方,每个主机都有相同的λ_i值
- ◆多跳路径，链路容量R
- ◆超时/重传

拥塞崩溃
R_2：λ'_{in}极大→BD流量
>>AC流量？→AC趋近于0



◆情境3



拥塞崩溃

拥塞的另一个”开销”：

- 当分组被丢弃时，该分组曾用到的所有“上游”传输容量被浪费了！

◆拥塞控制的方法

◆网络辅助的拥塞控制

- ◆直接网络反馈：路由器以阻塞分组的形式通知发送方“网络拥塞了”
- ◆经由接收方的网络反馈：路由器标识从发送方流向接收方分组中的某个字段以指示拥塞的产生，由接收方通知发送方“网络拥塞了”

◆端到端拥塞控制

- ◆网络层不为拥塞控制提供任何帮助和支持
- ◆端系统通过对网络行为（丢包或时延增加）的观测判断网络是否发生拥塞
- ◆目前TCP采用该种方法

3.7 TCP拥塞控制

- **TCP拥塞控制为端到端拥塞控制**

- **TCP进行拥塞控制的方法**
  - 每个发送方自动感知网络拥塞的程度
  - 发送方根据感知的结果限制外发的流量
    - 如果前方路径上出现了拥塞，则降低发送速率
    - 如果前方路径上没有出现拥塞，则增加发送速率

◆TCP拥塞控制需要解决的三个问题

◆TCP发送方如何限制外发流量的速率

- ◆ 拥塞窗口

$$\texttt{LastByteSent-LastByteAcked} \leq \texttt{CongWin}$$

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

◆发送方如何感知拥塞

- ◆ 超时
- ◆ 三个冗余ACK

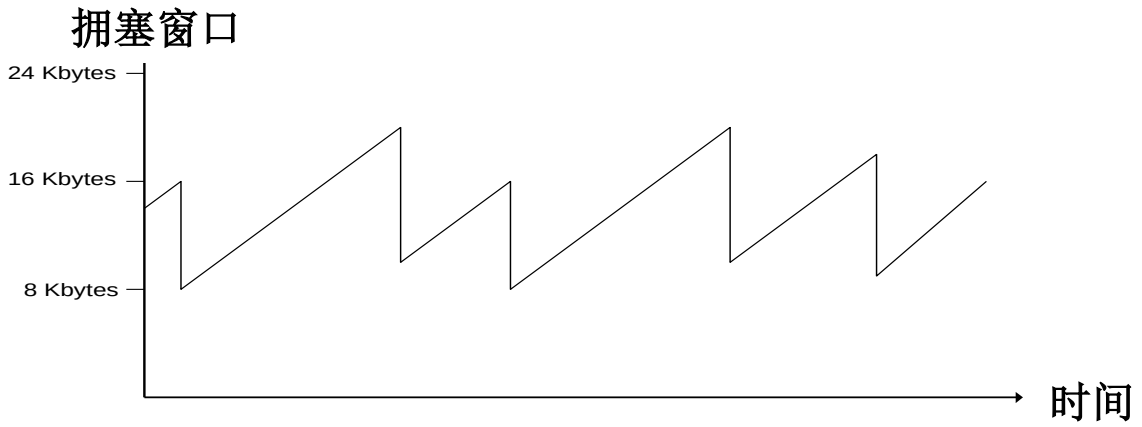
◆在感知到拥塞后，发送方如何调节发送速率

- ◆ 太快→拥塞崩溃
- ◆ 太慢→带宽利用率低

TCP拥塞控制算法（Reno算法）

加性增，乘性减（AIMD）

- 出现丢包事件后将当前 CongWin 大小减半，可以大大减少注入到网络中的分组数
- 当没有丢包事件发生了，每个RTT之后将CongWin增大1个MSS，使拥塞窗口缓慢增大，以防止网络过早出现拥塞



◆TCP拥塞控制算法（Reno算法）

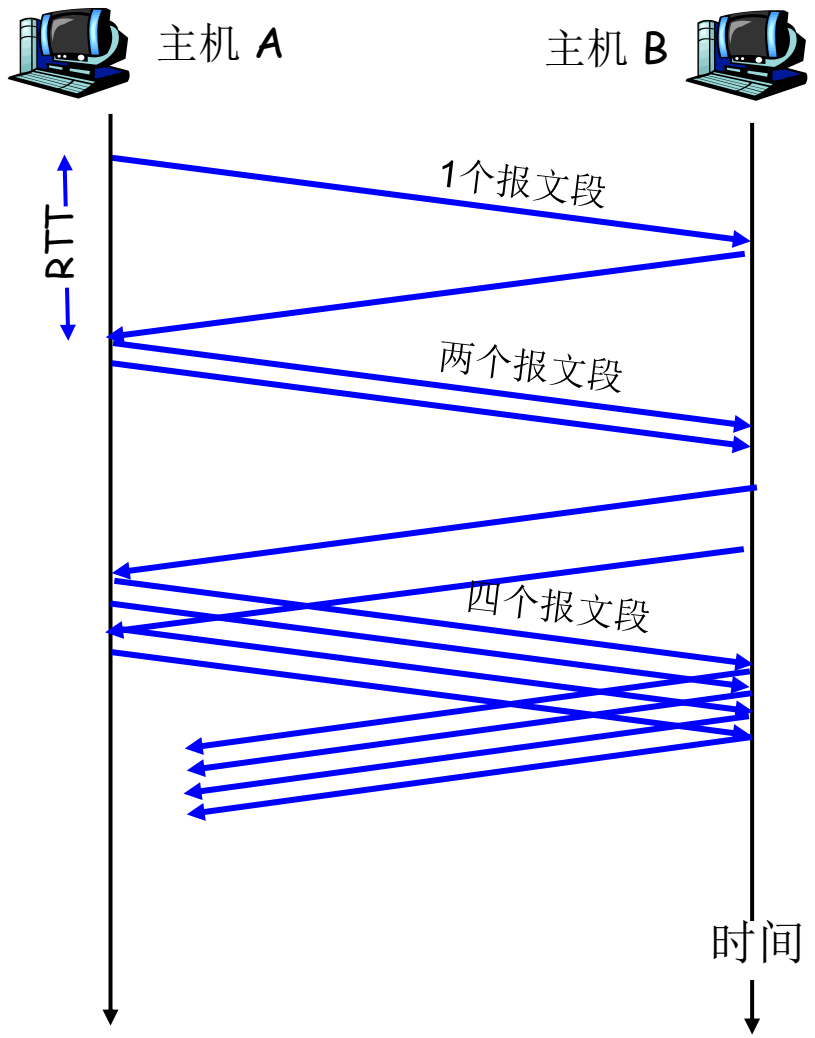
◆慢启动

- ◆建立连接时, **CongWin** = 1 MSS
  - ◆例如: MSS = 500 bytes & RTT = 200 msec
  - ◆初始速率 = 20 kbps
- ◆可用带宽 >> MSS/RTT
  - ◆初始阶段以指数的速度增加发送速率
- ◆连接初始阶段，以**指数的速度**增加发送速率，直到发生一个丢包事件为止
  - ◆每收到一次确认则将CongWin的值增加一个MSS

总结: 初始速率很低但速率的增长速度很快

3.7 TCP拥塞控制

◆慢启动

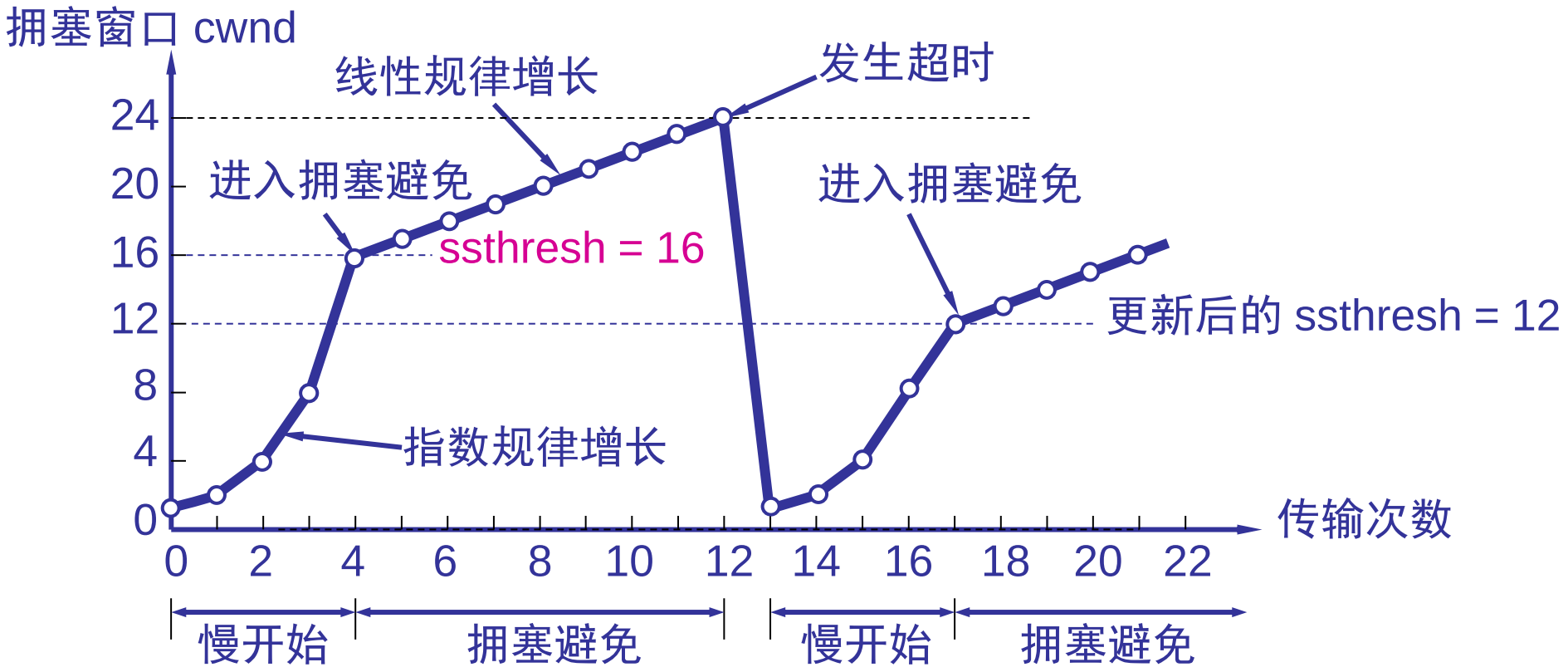


3.7 TCP拥塞控制

- 对收到**3个重复ACK**的反应
  - 将**CongWin减**为原来的一半
  - 线性增大拥塞窗口
- 对超时事件的反应
  - **门限值**设为当前CongWin的一半（门限值初始值65kB）
  - 将**CongWin**设为**1个 MSS**大小;
  - 窗口以指数速度增大
  - 窗口增大到门限值之后，再以线性速度增大

特别说明：早期的TCP Tahoe版本对上述两个事件并不区分，统一将CongWin降为1。实际上， 3个重复的ACK相对超时来说是一个预警信号，因此在Reno版中作了区分

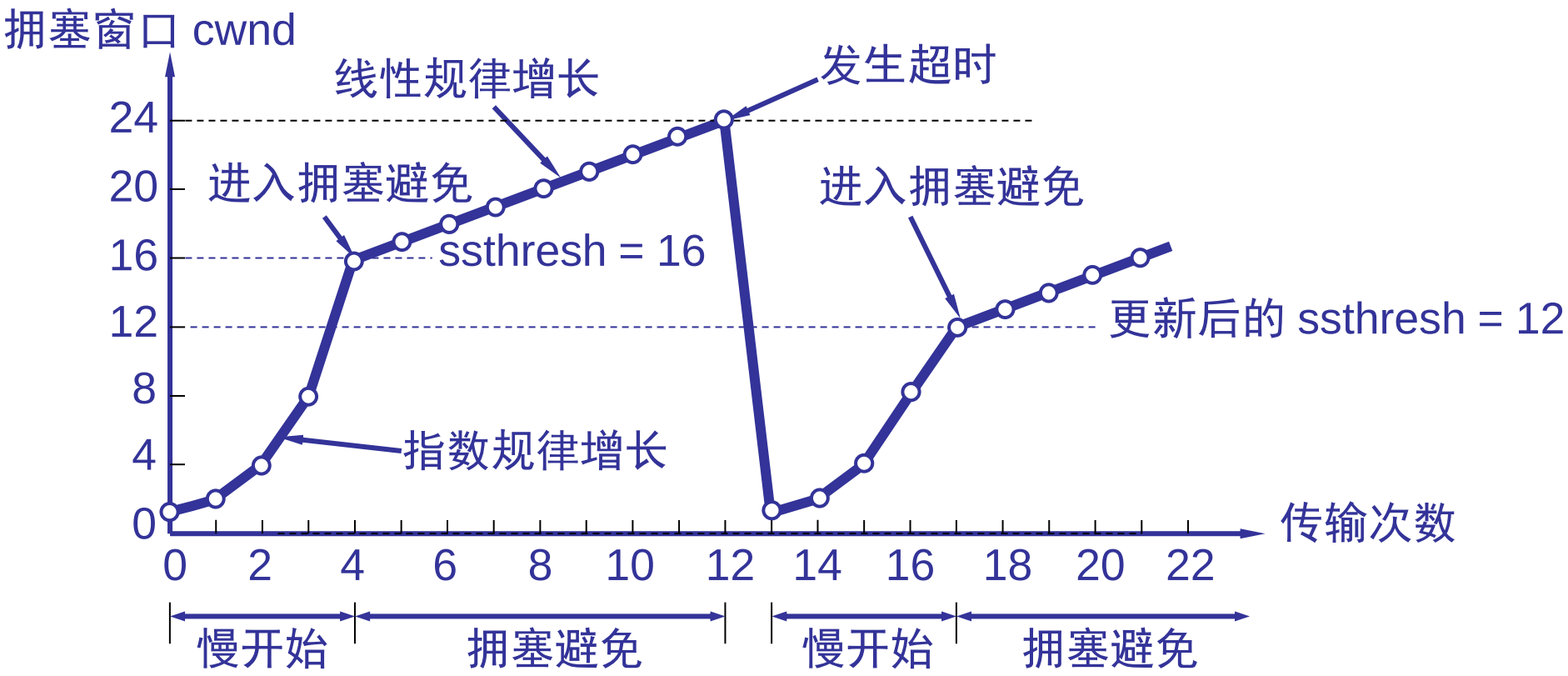
慢开始和拥塞避免算法的实现举例



当 TCP 连接进行初始化时，将拥塞窗口置为 1。图中的窗口单位不使用字节而使用报文段。

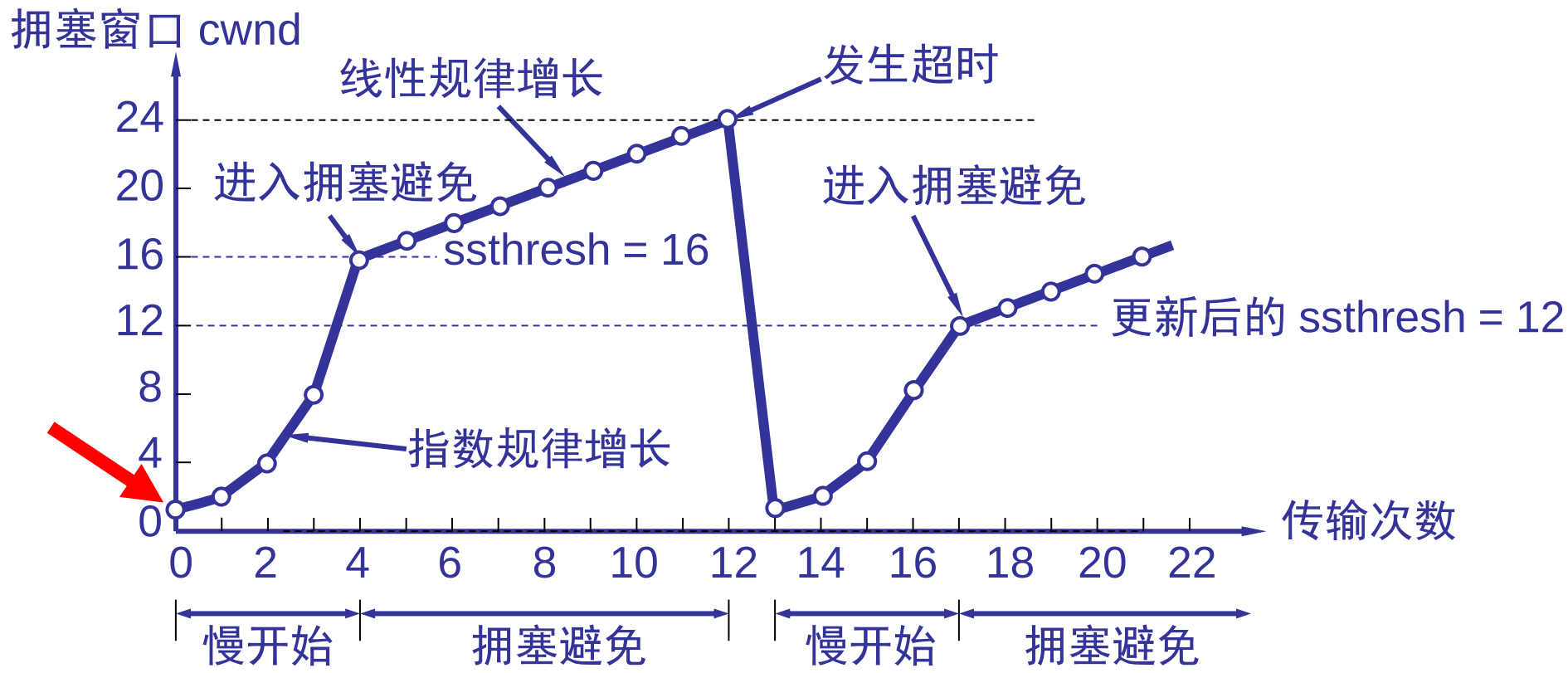
慢开始门限的初始值设置为 16 个报文段，即 ssthresh = 16。

慢开始和拥塞避免算法的实现举例



发送端的**发送窗口**不能超过**拥塞窗口** cwnd 和**接收端窗口** rwnd 中的**最小值**。我们假定接收端窗口足够大，因此现在发送窗口的数值等于拥塞窗口的数值。

慢开始和拥塞避免算法的实现举例



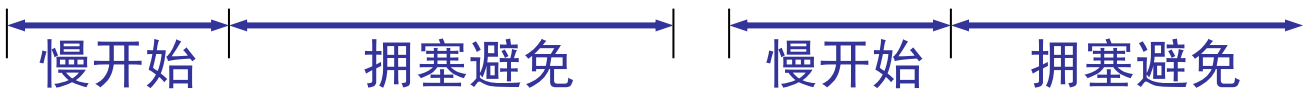
在执行慢开始算法时，拥塞窗口 cwnd 的初始值为 1，发送第一个报文段 M_0。

慢开始和拥塞避免算法的实现举例

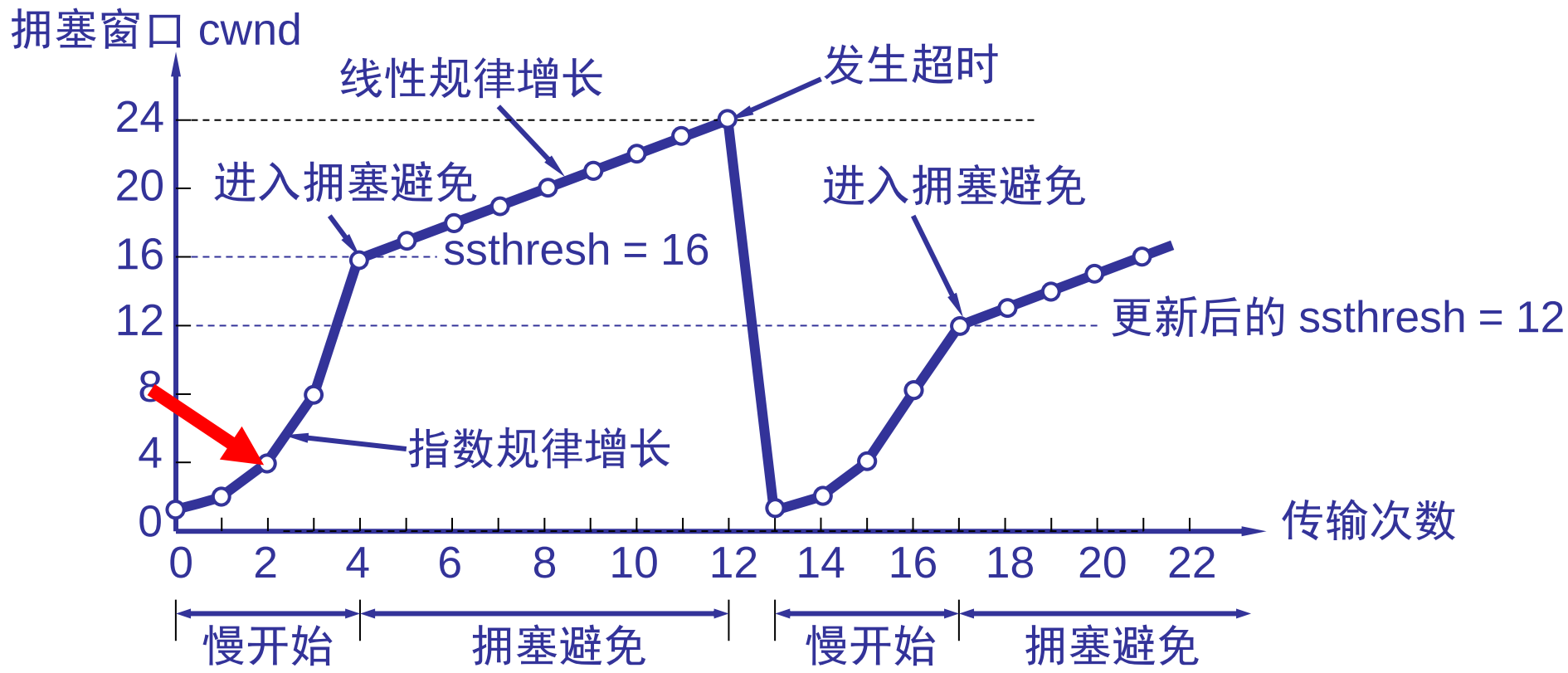
添加助教

12

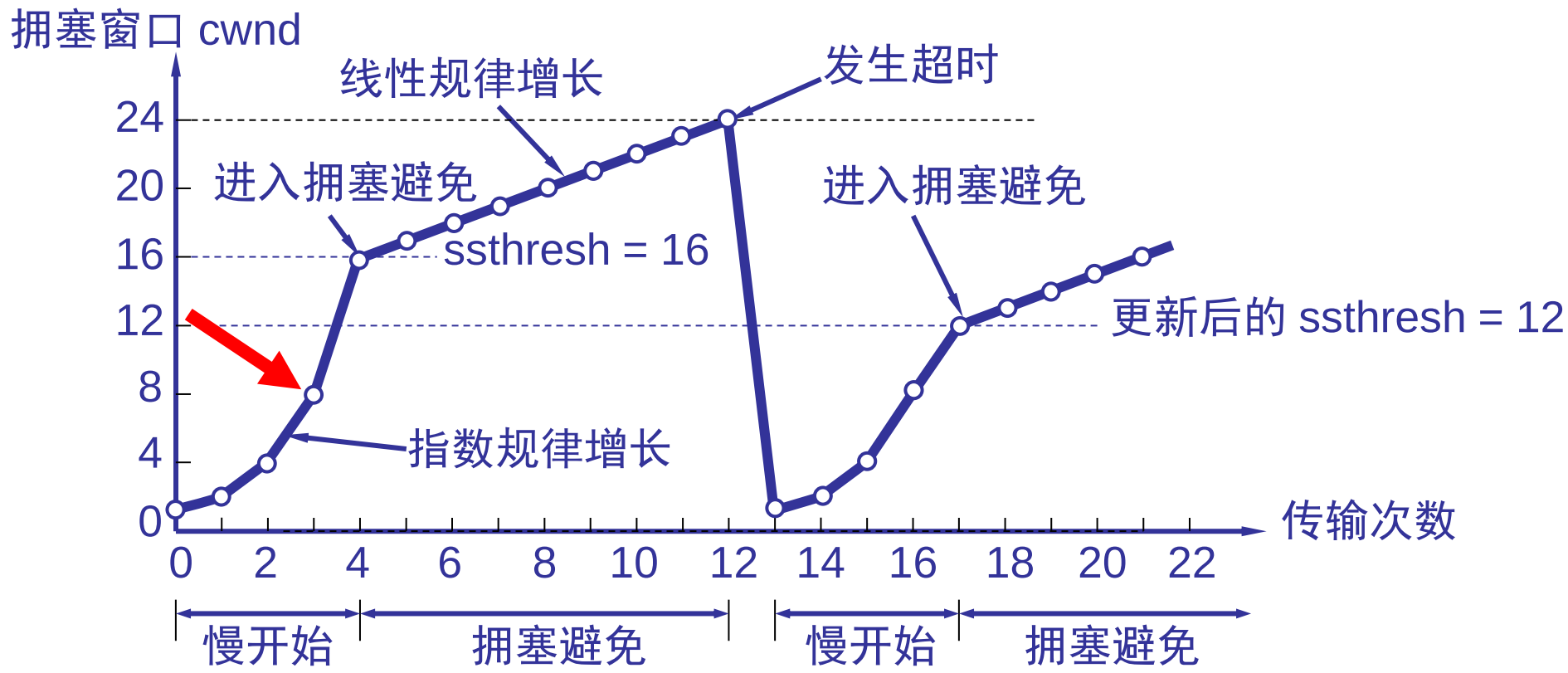
序号	助教	学号/工号	学校	加入时间	操作
☐	叶君妍	M202171777	华中科技大学	11-11	移除 查看
☐	王莉莉	20177840143	郑州大学	11-11	移除 查看
☐	马林威		超星网	11-15	移除 查看



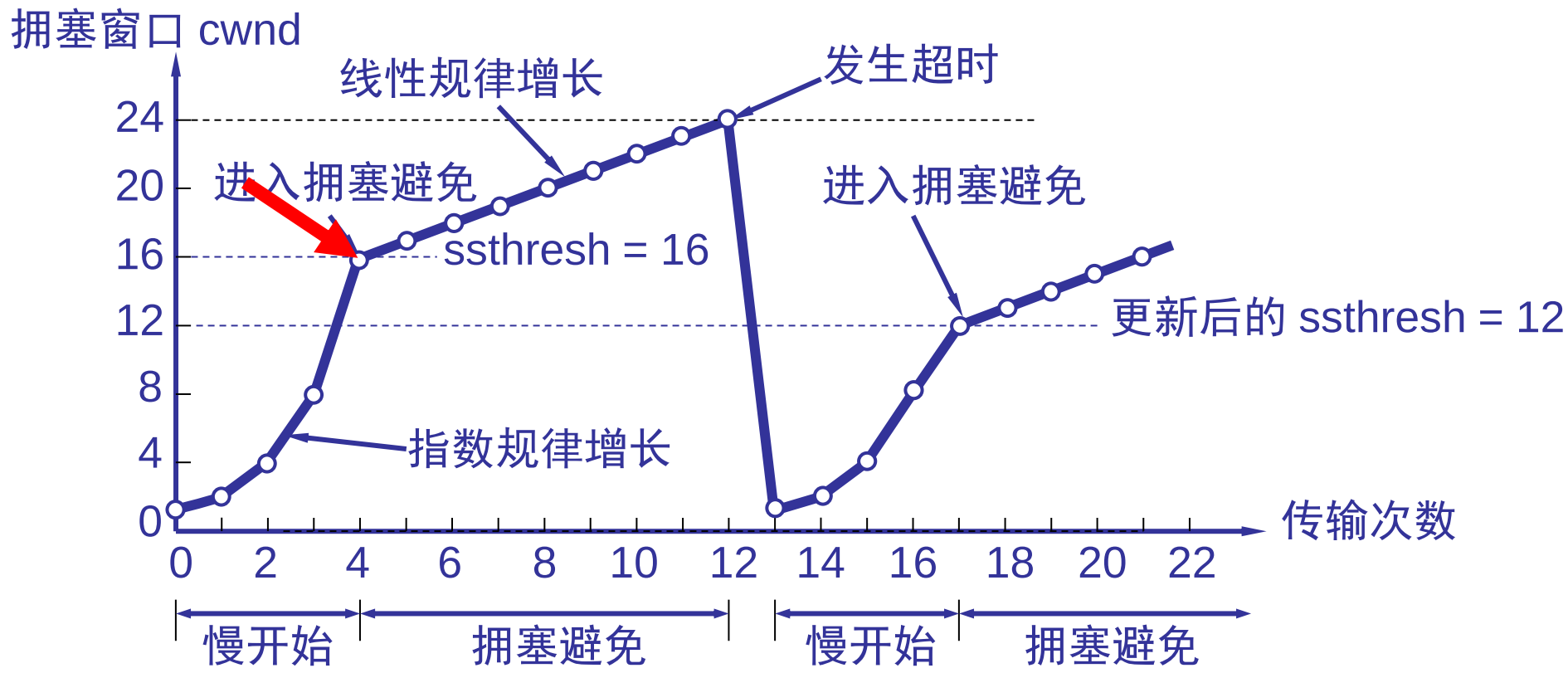
发送端收到 ACK_1（确认 M_0，期望收到 M_1）后，将 cwnd 从 1 增大到 2，于是发送端可以接着发送 M_1 和 M_2 两个报文段。



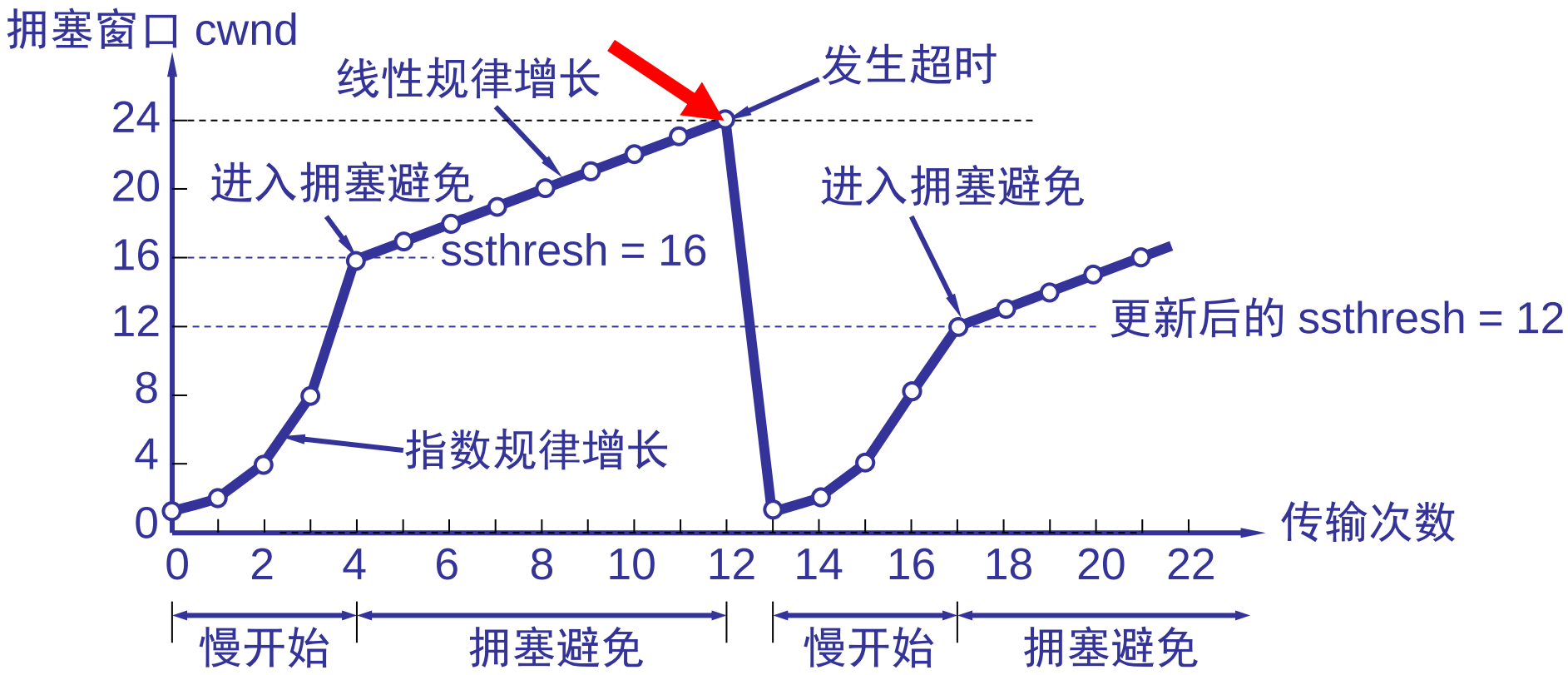
接收端发回 ACK_2 和 ACK_3。发送端每收到一个对新报文段的确认 ACK，就把发送端的拥塞窗口+1MSS。现在发送端的 cwnd 从 2 增大到 4，并可发送 M_4 ~ M_7共 4个报文段。



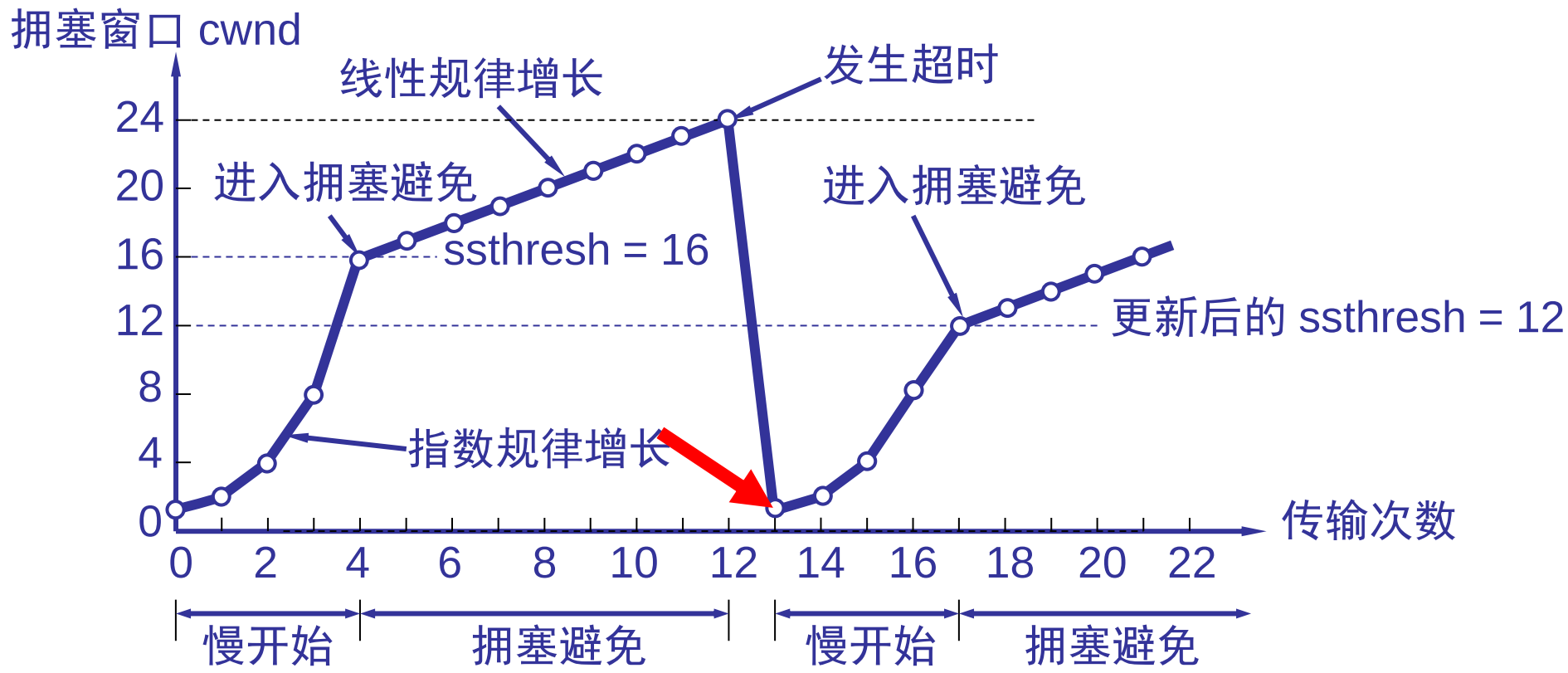
发送端每收到一个对新报文段的确认 ACK，就把发送端的拥塞窗口翻倍，因此拥塞窗口 cwnd 随着传输次数按指数规律增长。



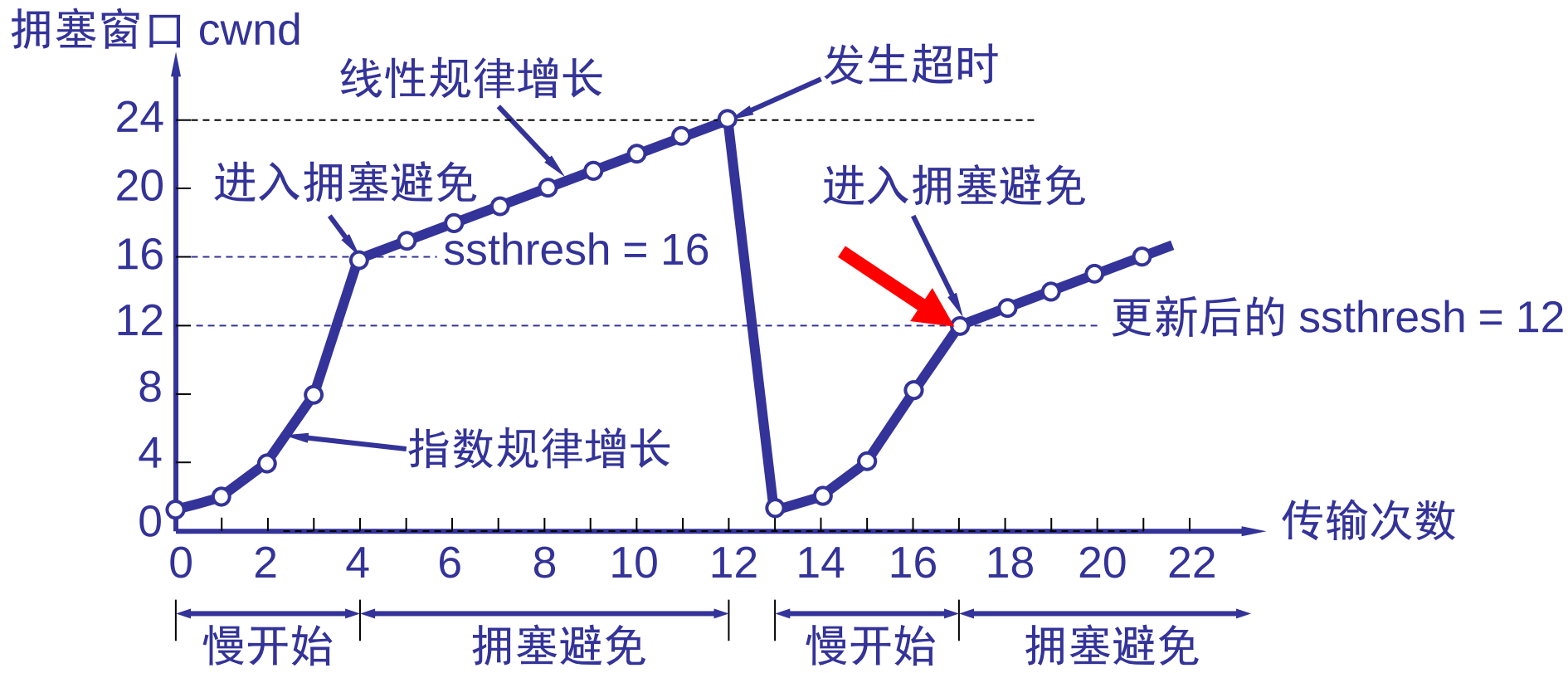
当拥塞窗口 cwnd 增长到慢开始门限值 ssthresh 时（即当 cwnd = 16 时），就改为执行拥塞避免算法，拥塞窗口按线性规律增长。



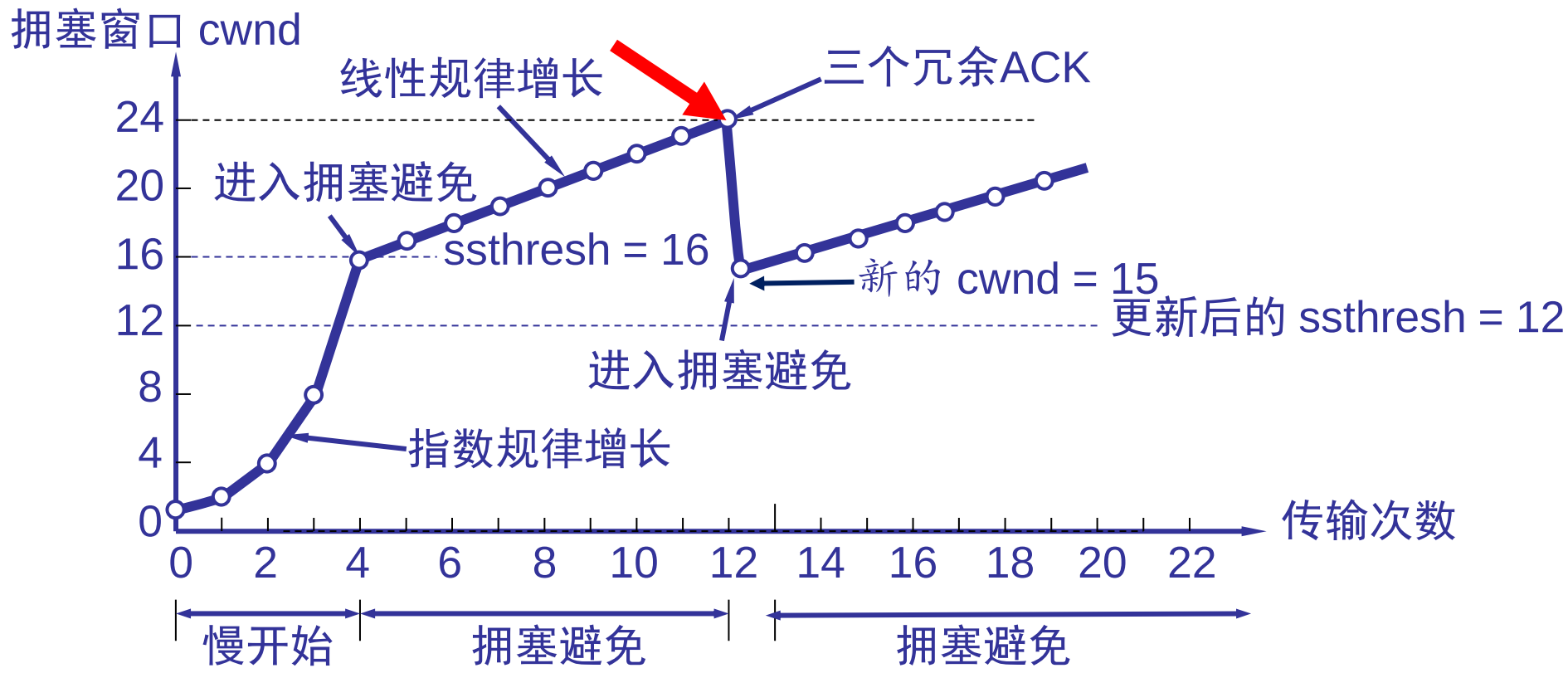
假定拥塞窗口的数值增长到 24 时，网络出现超时（表明网络拥塞了）。



更新后的 ssthresh 值变为 12（即发送窗口数值 24 的一半），
拥塞窗口再重新设置为 1，并执行慢开始算法。



当 cwnd = 12 时改为执行拥塞避免算法，拥塞窗口按线性规律增长，每经过一个往返时延就增加一个 MSS 的大小。



假定拥塞窗口的数值增长到 24 时，网络出现冗余ACK

- 快速恢复（TCP推荐但非必须实现）
  - 3个冗余ACK进入快速重传后
  - 每收到一个冗余ACK：CongWin++
  - 直至收到一个新的ACK：CongWin=门限值，重新进入拥塞避免
  - 在进入快速恢复之后及重新进入拥塞避免之间，如果出现超时现象，直接按照前述超时事件进行处理

说明：本页内容不纳入考试范围

3.7 TCP拥塞控制

◆额外说明

- ◆快速恢复和超时中，门限值并不总等于CongWin/2
- ◆门限值=Max(flightSize/2，2MSS)
  - ◆ flightsize:当时发送窗口中已发出但未确认的报文段数目
- ◆门限值=Max(min(拥塞窗口,通知窗口)，2MSS)　–微软

说明：本页内容不纳入考试范围

◆TCP拥塞控制算法（Reno）总结

- ◆当 拥塞窗口CongWin小于门限值Threshold时，发送方处于 慢启动 阶段，窗口以指数速度增大。
- ◆当 拥塞窗口CongWin大于门限值Threshold时，发送方处于 拥塞避免 阶段，窗口线性增大。
- ◆当收到 3个重复的ACK 时,门限值Threshold设为拥塞窗口的1/2，而拥塞窗口CongWin设为门限值Threshold+3MSS ，收到新的ACK，则拥塞窗口CongWin设为门限值Threshold。
- ◆当 超时 事件发生时，门限值Threshold设为拥塞窗口的1/2，而拥塞窗口CongWin设为1个 MSS。

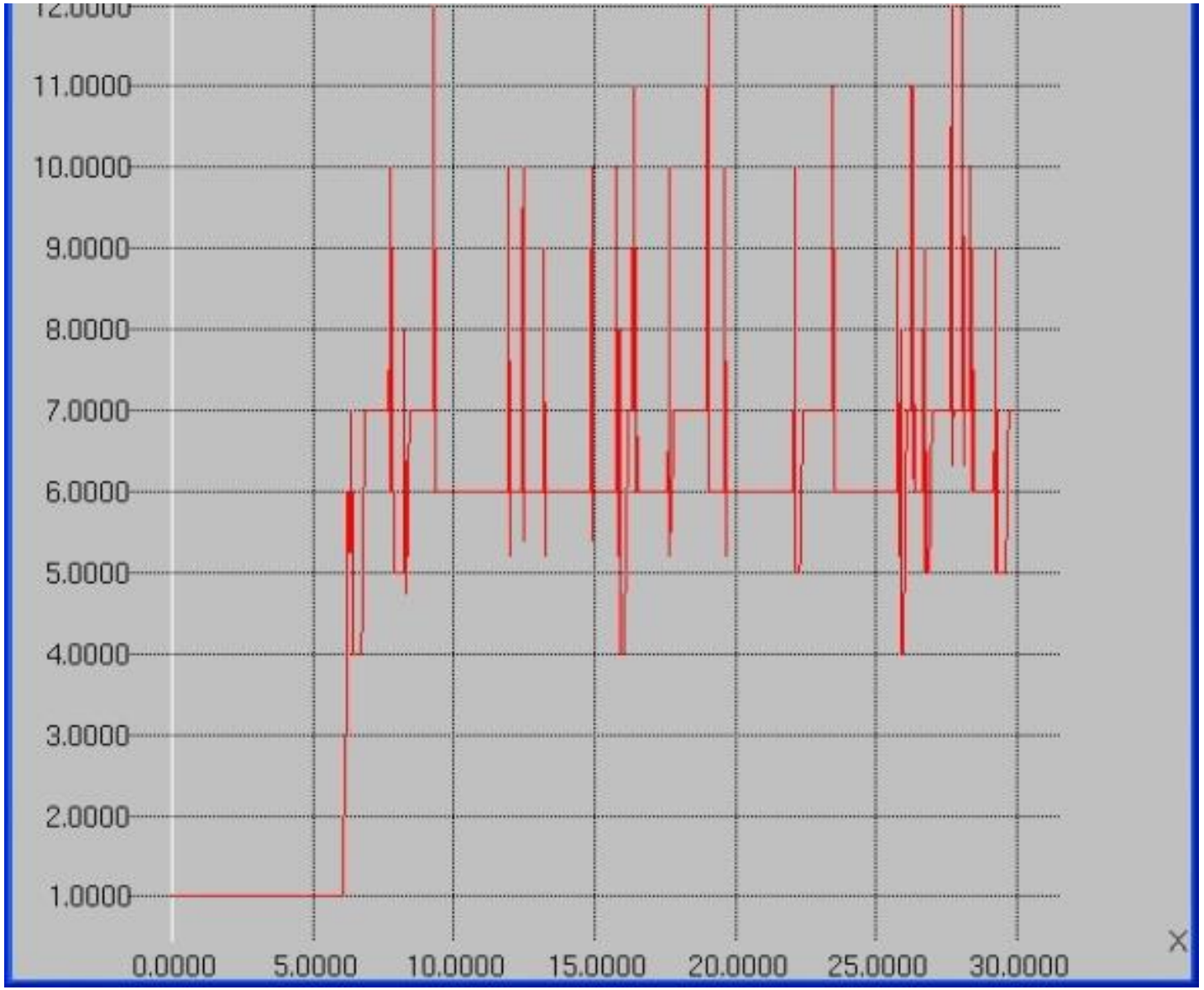
3.7 TCP拥塞控制

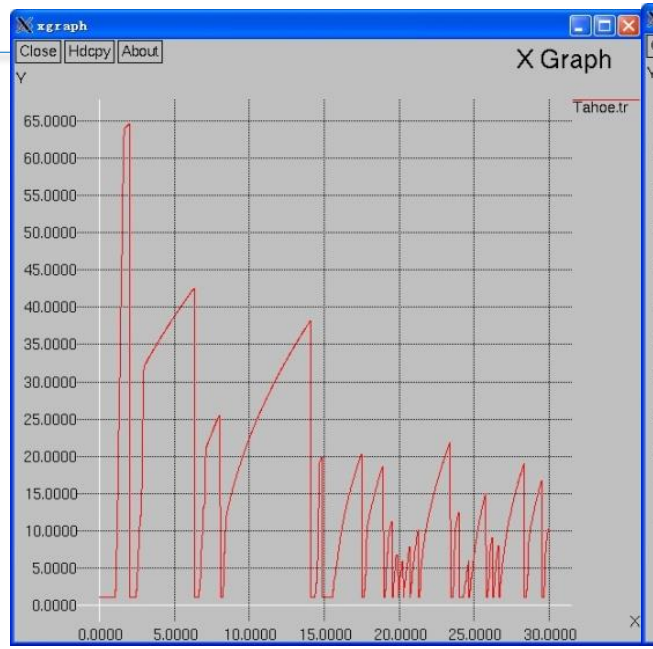
事件	状态	TCP发送方动作	说明
收到前面未确认数据的ACK	慢启动(SS)	CongWin = CongWin + MSS, If (CongWin > Threshold) 设置状态为 “拥塞避免”	导致每过一个RTT则CongWin翻倍
收到前面未确认数据的ACK	拥塞避免(CA)	CongWin = CongWin+MSS * (MSS/CongWin)	加性增，每个RTT导致CongWin 增大1个 MSS
由3个重复ACK检测到丢包事件	SS 或 CA	Threshold = CongWin/2, CongWin = Threshold, 设置状态为 “拥塞避免”	快速恢复，实现乘性减. CongWin不低于1个MSS.
超时	SS 或 CA	Threshold = CongWin/2, CongWin = 1 MSS, 设置状态为 “慢启动”	进入慢启动
重复ACK	SS 或 CA	对确认的报文段增加重复ACK的计数	CongWin 和Threshold不变

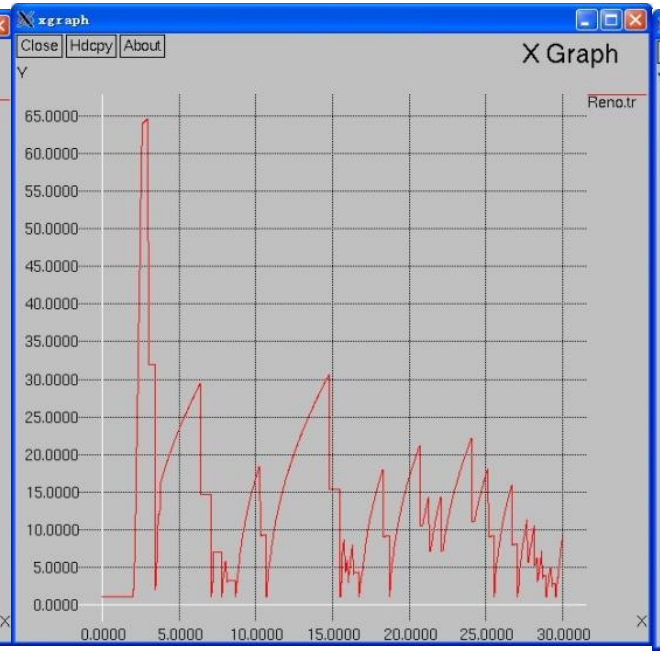
◆Reno算法的演进——Vegas算法

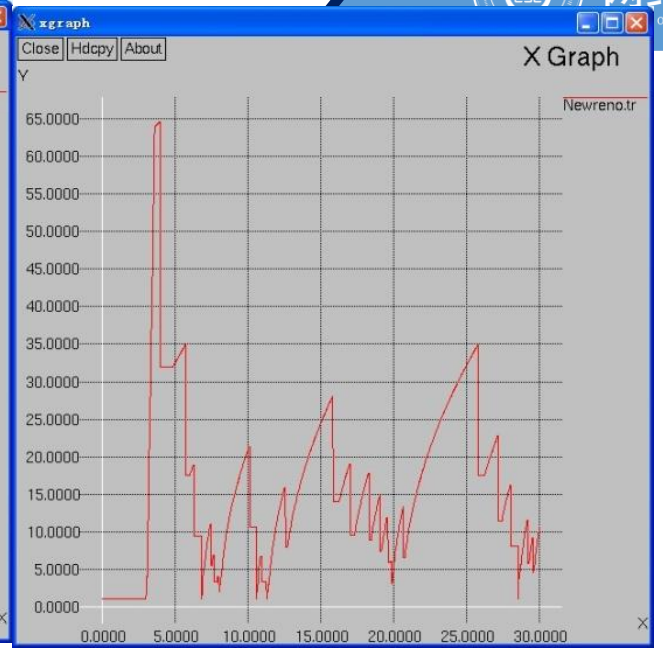
- ◆通过往返时延的变化检测拥塞的严重程度
  - ◆ 往返时延越长，拥塞越严重
- ◆当检测的拥塞达到一定程度时，有意识的线性降低发送速率以避免拥塞

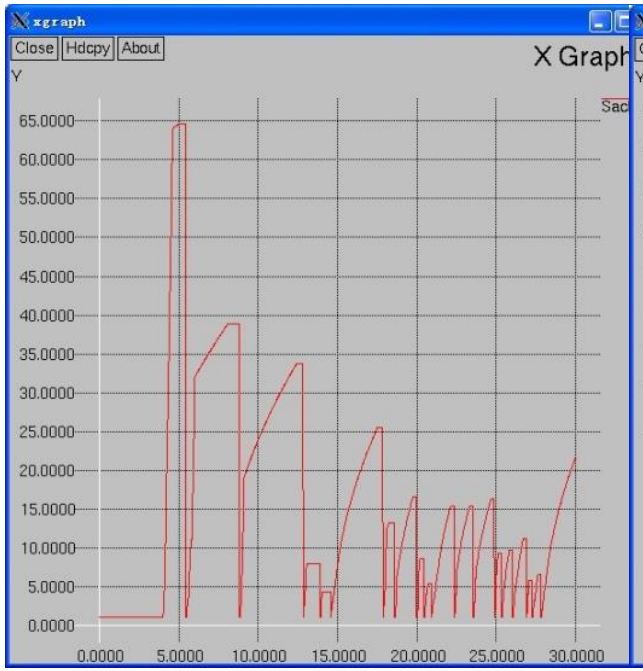
说明：本页内容不纳入考试范围

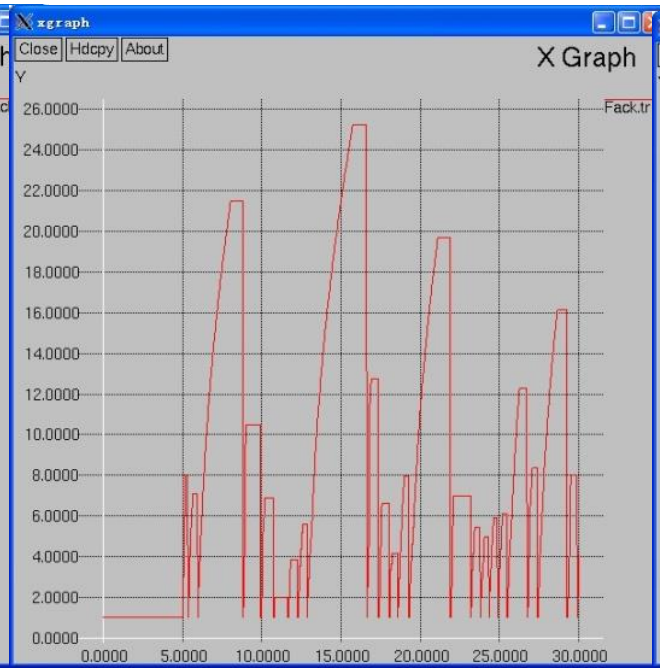

12.0000
11.0000
10.0000
9.0000
8.0000
7.0000
6.0000
5.0000
4.0000
3.0000
2.0000
1.0000
0.0000
5.0000
10.0000
15.0000
20.0000
25.0000
30.0000

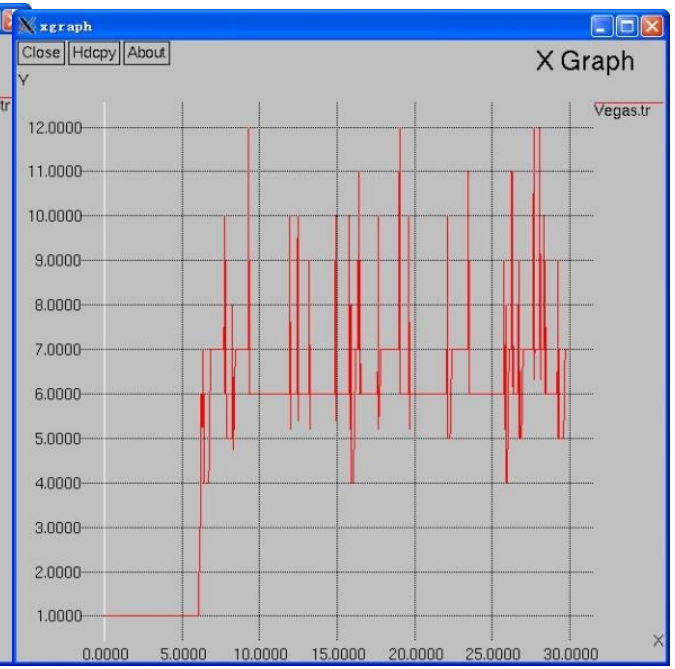

xgraph
Close Hdcpy About
X Graph
Y
Tahoe.tr
65.0000
60.0000
55.0000
50.0000
45.0000
40.0000
35.0000
30.0000
25.0000
20.0000
15.0000
10.0000
5.0000
0.0000
0.0000 5.0000 10.0000 15.0000 20.0000 25.0000 30.0000
X


xgraph
Close Hdcpy About
X Graph
Y
Reno.tr
65.0000
60.0000
55.0000
50.0000
45.0000
40.0000
35.0000
30.0000
25.0000
20.0000
15.0000
10.0000
5.0000
0.0000
0.0000 5.0000 10.0000 15.0000 20.0000 25.0000 30.0000
X


xgraph
Close Hdcpy About
X Graph
Y
Newreno.tr
65.0000
60.0000
55.0000
50.0000
45.0000
40.0000
35.0000
30.0000
25.0000
20.0000
15.0000
10.0000
5.0000
0.0000
0.0000 5.0000 10.0000 15.0000 20.0000 25.0000 30.0000
X


xgraph
Close Hdcpy About
X Graph
Y
65.0000
60.0000
55.0000
50.0000
45.0000
40.0000
35.0000
30.0000
25.0000
20.0000
15.0000
10.0000
5.0000
0.0000
0.0000 5.0000 10.0000 15.0000 20.0000 25.0000 30.0000


xgraph
Close Hdcpy About
X Graph
Y
Fack.tr
26.0000
24.0000
22.0000
20.0000
18.0000
16.0000
14.0000
12.0000
10.0000
8.0000
6.0000
4.0000
2.0000
0.0000
0.0000 5.0000 10.0000 15.0000 20.0000 25.0000 30.0000


xgraph
Close Hdcpy About
X Graph
Y
Vegas.tr
12.0000
11.0000
10.0000
9.0000
8.0000
7.0000
6.0000
5.0000
4.0000
3.0000
2.0000
1.0000
0.0000 5.0000 10.0000 15.0000 20.0000 25.0000 30.0000
X

◆TCP的吞吐量

- ◆作为窗口大小和RTT的函数TCP的平均吞吐量应该是什么样的?
  - ◆ 忽略慢启动
- ◆假定当丢包事件发生时，窗口大小为 W.
  - ◆ 此时 吞吐量为W/RTT
- ◆丢包事件发生后，窗口大小减为W/2, 吞吐量为W/2RTT.
- ◆因此平均吞吐量为: 0.75 W/RTT

◆TCP吞吐量的进一步讨论

- ◆吞吐量是丢包率(L)的函数:

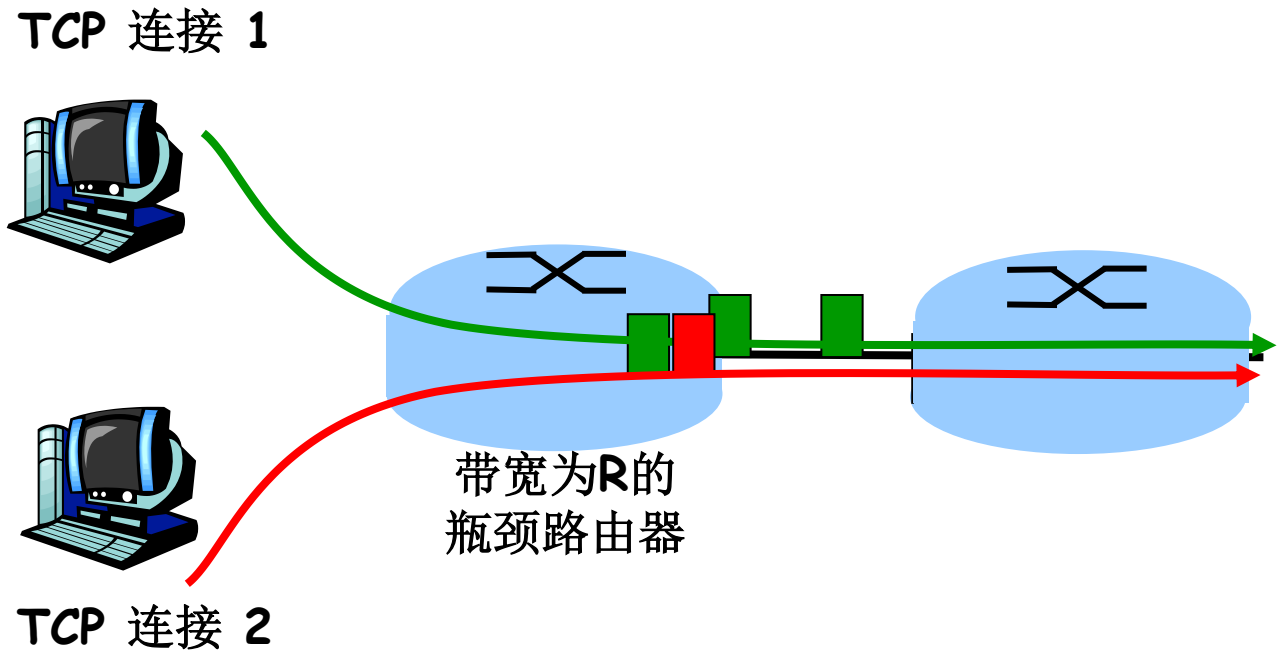
$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ◆对于一条MSS=1500字节，RTT=100ms的TCP连接而言，如果希望达到10Gbps的吞吐量，那么丢包率L不能高于2x10-10

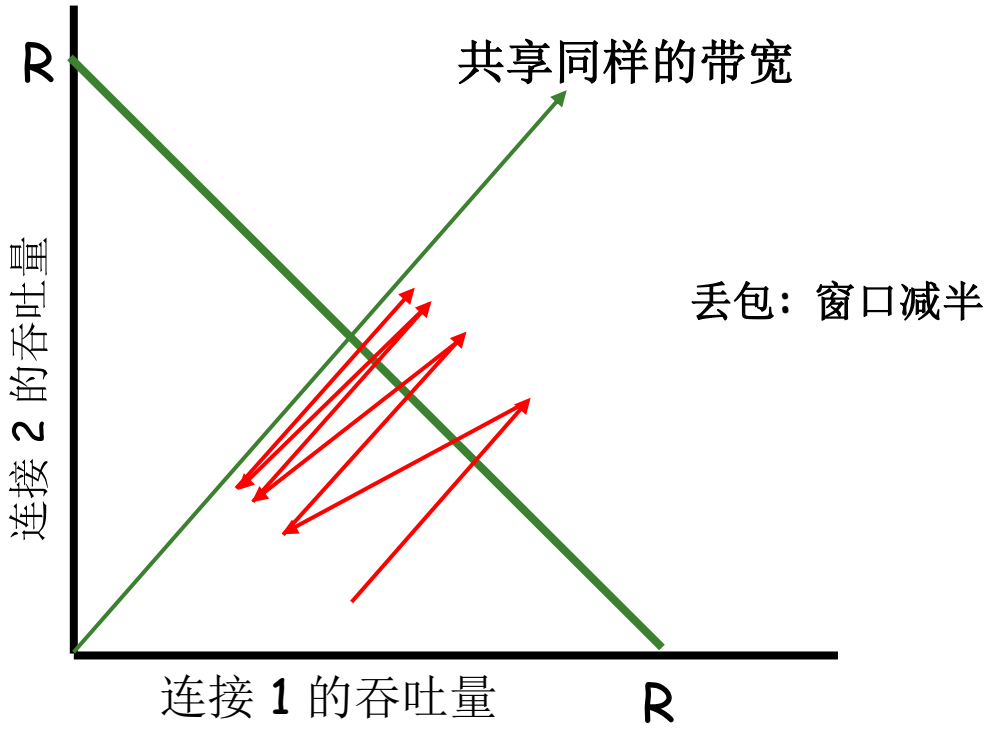
◆TCP拥塞控制的公平性分析

◆公平性的目标

- ◆ 如果K个TCP连接共享同一个带宽为R的瓶颈链路, 每个连接的平均传输速率为 R/K



◆TCP的公平性



3.7 TCP拥塞控制

- 公平性和UDP
  - 多媒体应用一般不使用TCP
    - 不希望因为拥塞控制影响其速率
  - 多媒体应用采用UDP:
    - 恒定的速率传输音频和视频数据，可容忍丢包
- 公平性和并行TCP连接
  - 无法阻止应用在两个主机之间建立多个并行的连接.
  - Web浏览器就是这样
  - 例子: 速率为R 的链路当前支持9个并发连接;
    - 应用请求一个TCP连接，获得R/10的速率
    - 应用请求11个TCP连接，获得R/2的速率!

作 业

- 习题
  - 18、27、31、40、45、56