计算机网络

TCP

苏铅坤

理想传输

- 理想的传输条件有以下两个特点：
  - 传输信道不产生差错。
  - 不管发送方以多快的速度发送数据，接收方总是来得及处理收到的数据

- 然而实际的网络都不具备以上两个理想条件。必须使用一些可靠传输协议，在不可靠的传输信道实现可靠传输

目录
CONTENTS

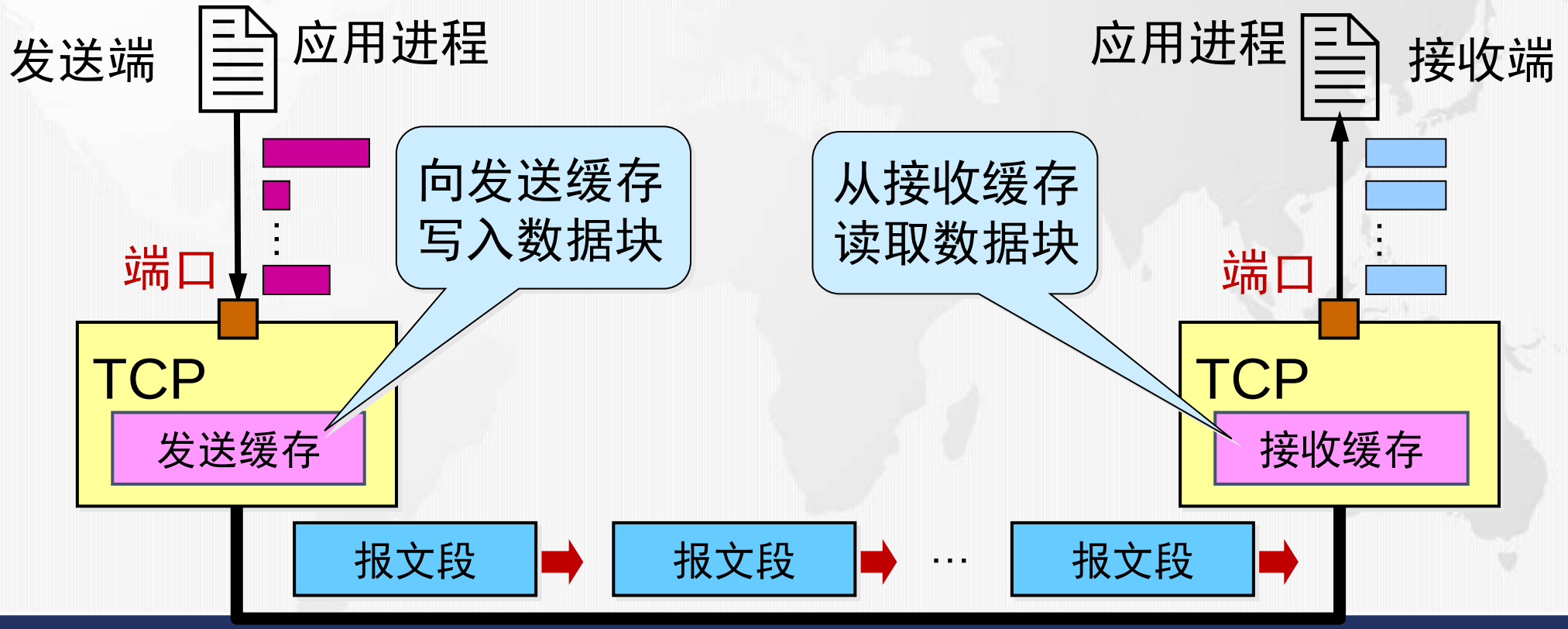
报文太大，怎么办？

合并与拆分

- 面向字节流
  - 把应用程序交下来的数据看成是一连串无结构的字节流（字节系列）
- 收到的字节流必须和发出的字节流完全一样
  - 报文未必按序到达

对字节流进行分段

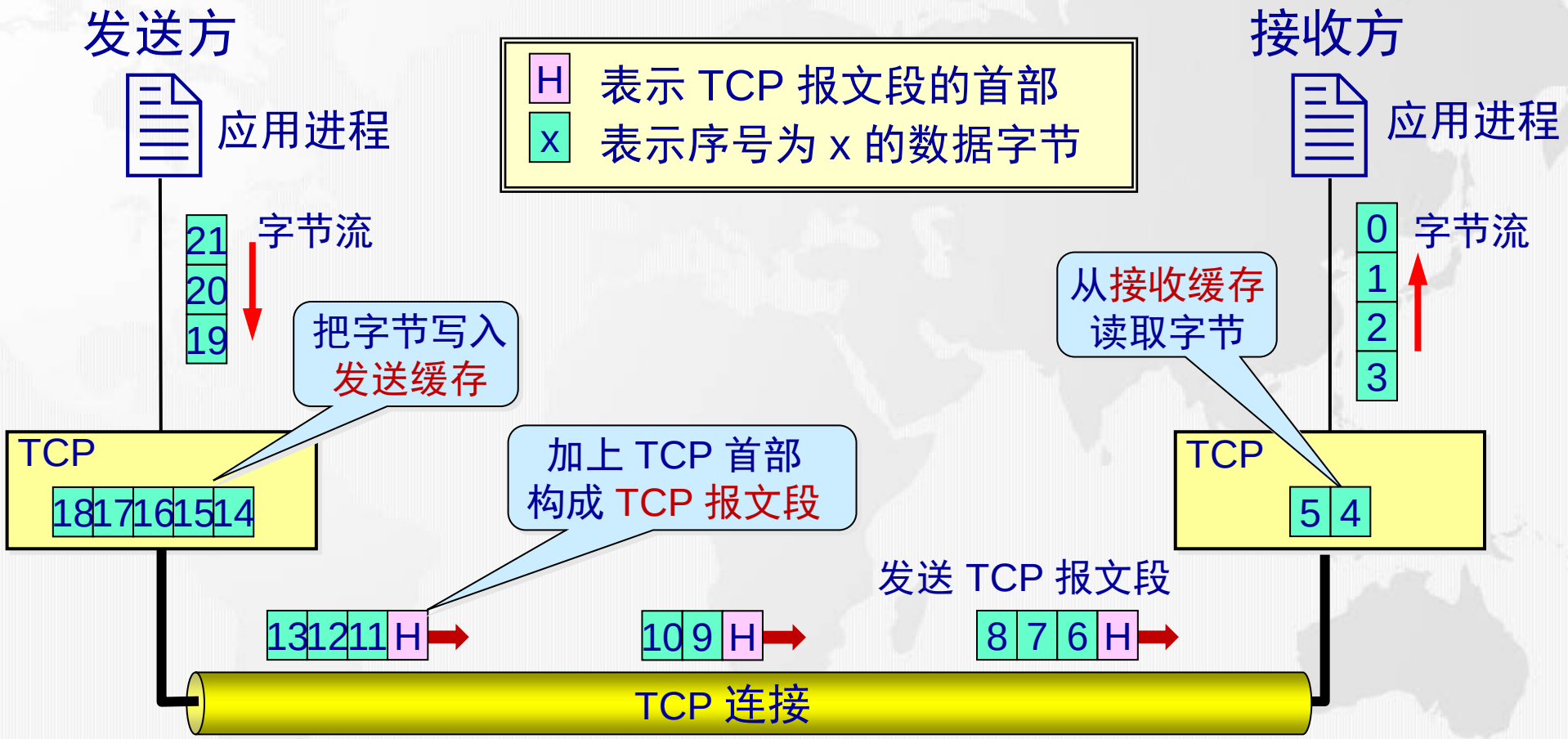
- TCP不关心应用进程一次把多长的报文发送到TCP缓存
  - TCP对连续的字节流进行分段，形成TCP报文段



合并与拆分

- TCP把太长的数据块划分短一些再传送
  - TCP根据对方给出的窗口值和当前网络拥塞的程度来决定一个报文段应包含多少个字节
- TCP也可等待积累有足够多的字节后再构成报文段发送出去

面向字节流

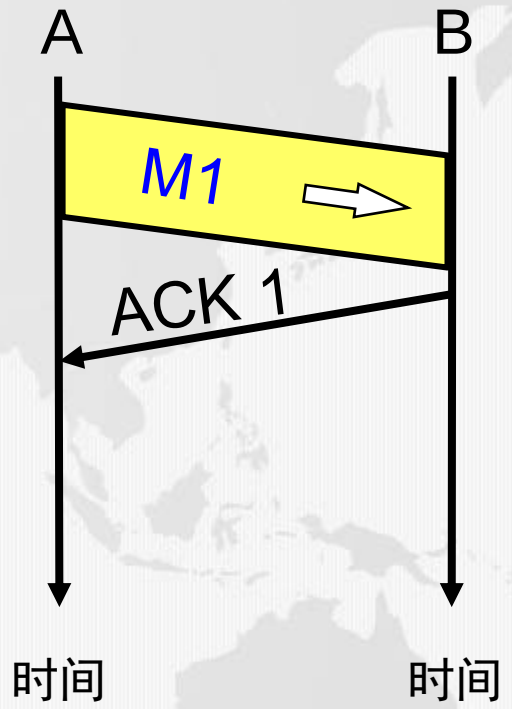

发送方
应用进程
H 表示 TCP 报文段的首部
x 表示序号为 x 的数据字节
接收方
应用进程
21
20
19
字节流
把字节写入
发送缓存
TCP
18 17 16 15 14
加上 TCP 首部
构成 TCP 报文段
13 12 11 H
10 9 H
发送 TCP 报文段
8 7 6 H
从接收缓存
读取字节
0
1
2
3
字节流
TCP
5 4
TCP 连接

目录
CONTENTS

怎么做到可靠传输?

确认机制

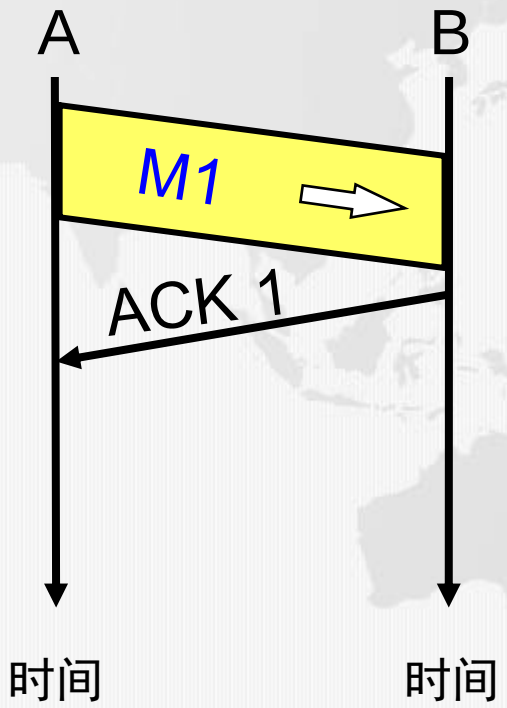
☺ 确认机制



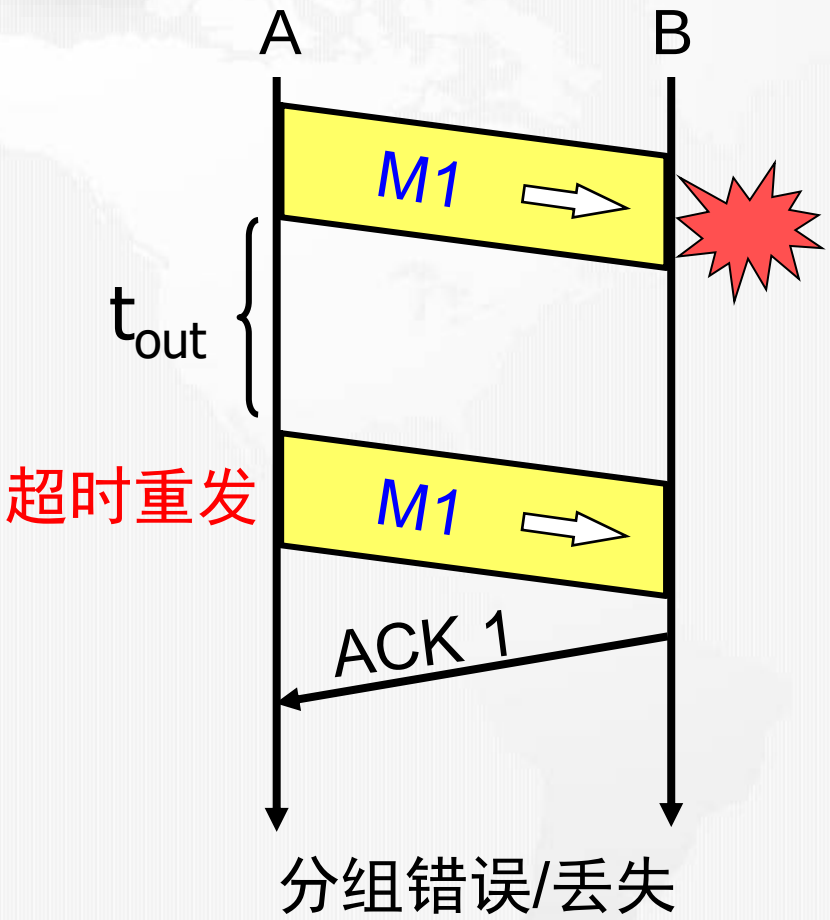
这个过程会出现哪些差错?

出现差错

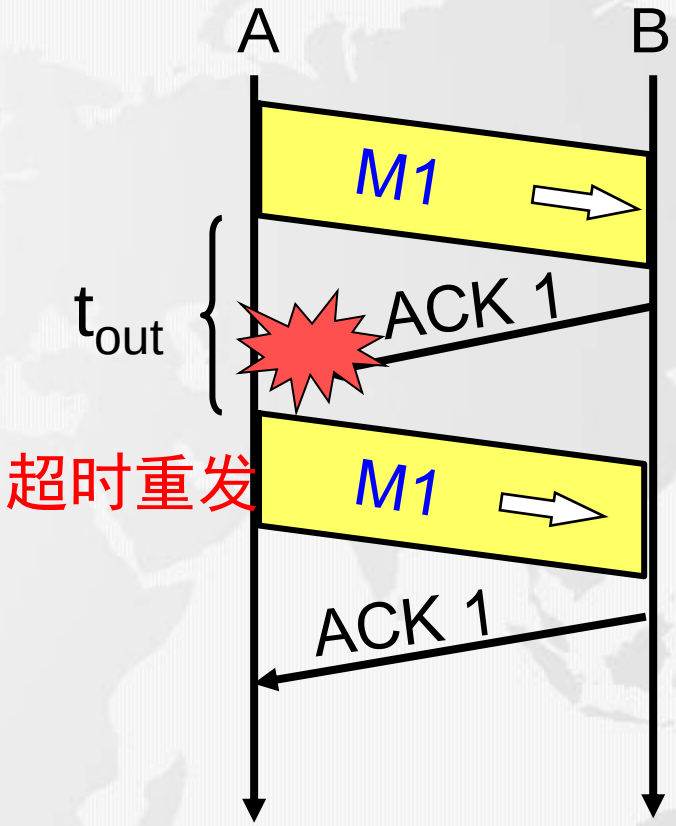
- M1在传输过程中丢失了，B什么都不知道，也什么都不做
- B接收M1时检测出了差错
- 确认丢失
- 确认出错



超时重传



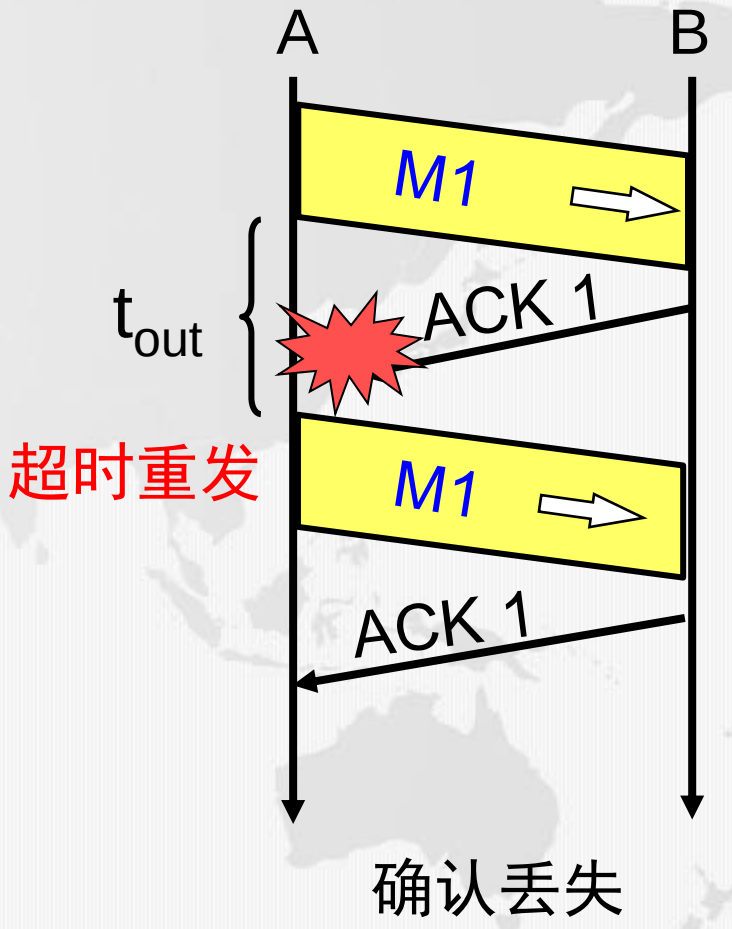
分组错误/丢失



确认出错/丢失

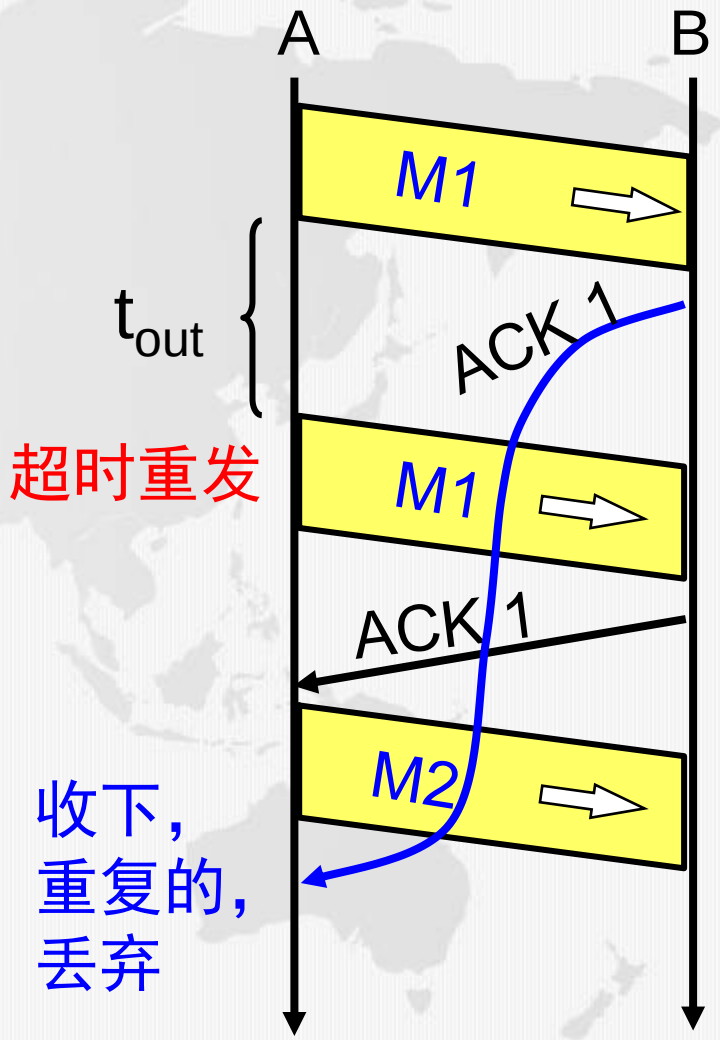
确认丢失，收到两次分组

- ☺ 假定B又收到了重传的分组 M1，B 应采取两个行动：
  - 丢弃这个重复的分组 M1，不向上层交付
  - 向 A 发送确认。不能认为已经发送过确认就不再发送，因为 A 之所以重传 M1 就表示 A 没有收到对 M1 的确认



确认迟到

- ☺ 传输过程中没有出现差错，但 B对分组 M1的确认迟到了
  - A会收到重复的确认。对重复的确认的处理很简单：收下后就丢弃。
  - B仍然会收到重复的 M1，并且同样要丢弃重复的 M1，并重传确认分组。



自动重传请求ARQ

- 上述确认和重传机制，可以在不可靠的传输网络上实现可靠的通信

- 像上述的这种可靠传输协议常称为**自动重传请求ARQ** (Automatic Repeat reQuest)
  - 重传的请求是自动进行的，接收方不需要请求发送方重传某个出错的分组

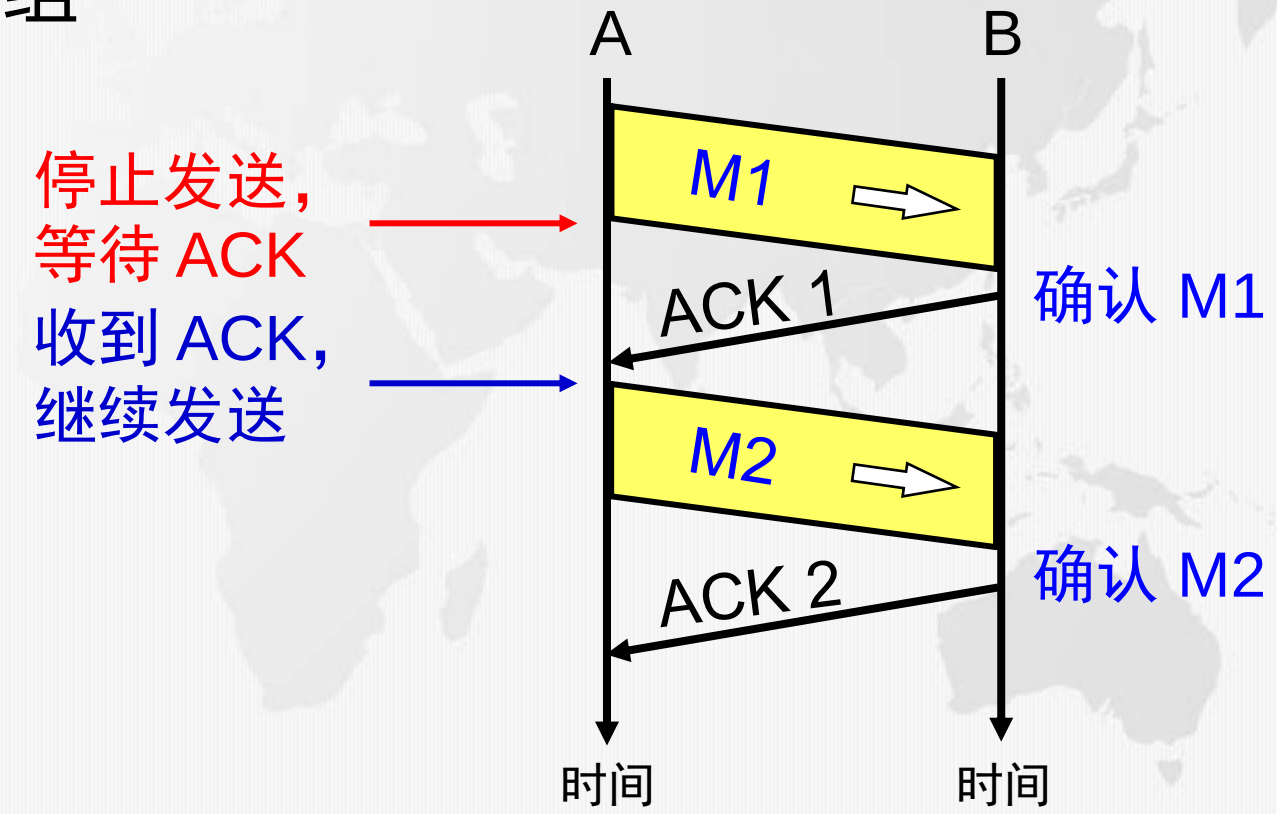
还有

- 在发送完一个分组后，必须暂时保留已发送的分组的副本，以备重发
- 通常A最终总是可以收到对所有发出的分组的确认
  - 如果A不断重传分组但总是收不到确认，就说明通信线路太差，不能进行通信
- 分组和确认分组都必须进行编号

2.1 停止等待协议

停止等待协议

☺ “停止等待”就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组

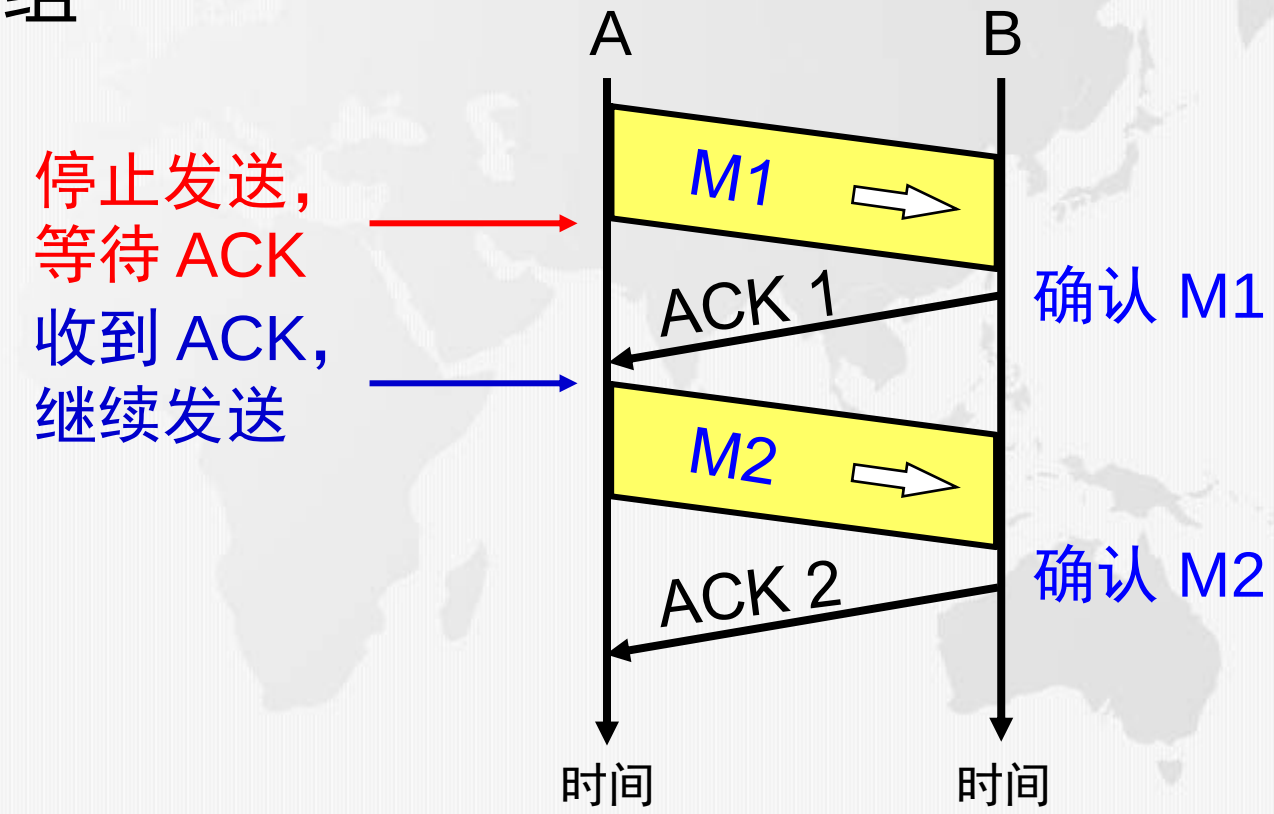


停止等待协议有什么缺点?

2.2 停止等待协议缺点

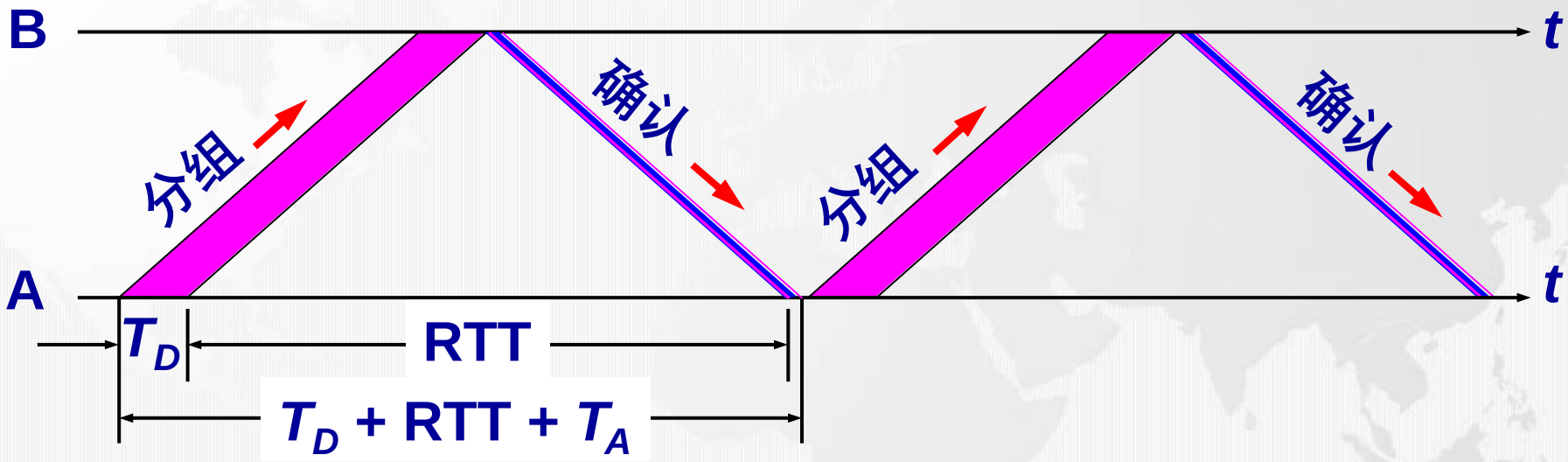
停止等待协议

☺ “停止等待”就是每发送完一个分组就停止发送，等待对方的确认。在收到确认后再发送下一个分组



信道利用率低

☺ 停止等待协议的优点是简单，缺点是信道利用率太低



信道利用率 $U = \dfrac{T_D}{T_D + \mathrm{RTT} + T_A}$

练习：求信道利用率

☺ 假设信道利用率U定义如下（假设A处理确认分组时间忽略不计）：

$$\text{信道利用率 } U = \frac{T_D}{T_D + \mathrm{RTT} + T_A}$$

☺ 其中：

- Td：A发送分组需要的时间
- RTT：往返时间
- Ta：B发送确认分组的时间

☺ 若RTT=19 ms, Ta = 0, 分组长度为1000 bit, 发送速率是1Mbit/s，那么信道利用率等于多少？ (1M = 1000K, 1K=1000)

重传，信道利用率会更低

信道利用率 $U = \dfrac{T_D}{T_D + \mathrm{RTT} + T_A}$

- ☺ 当往返时间RTT远大于分组发送时间T_D时，信道的利用率就会非常低
- ☺ 若出现重传，则对传送有用的数据信息来说，信道的利用率就还要降低

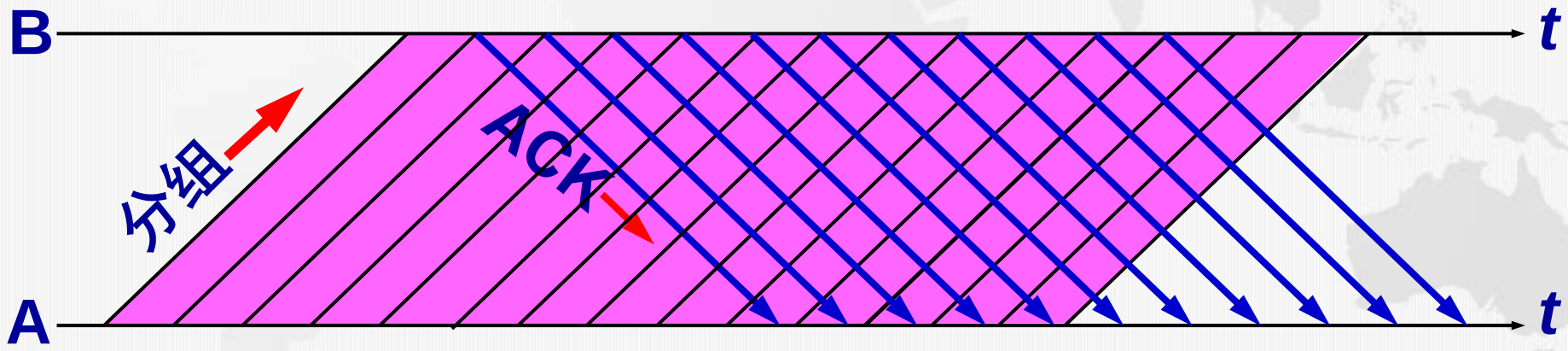
如何提升信道利用率?

2.3 流水线传输

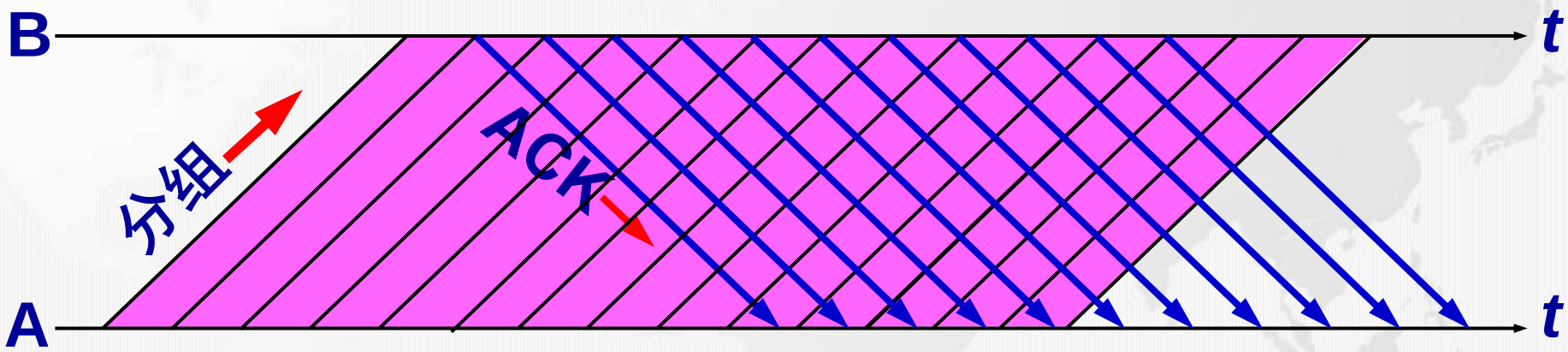
流水线传输

☺ 流水线传输就是发送方可连续发送多个分组，不必每发完一个分组就停顿下来等待对方的确认

信道利用率 $U = \dfrac{T_D}{T_D + \mathrm{RTT} + T_A}$

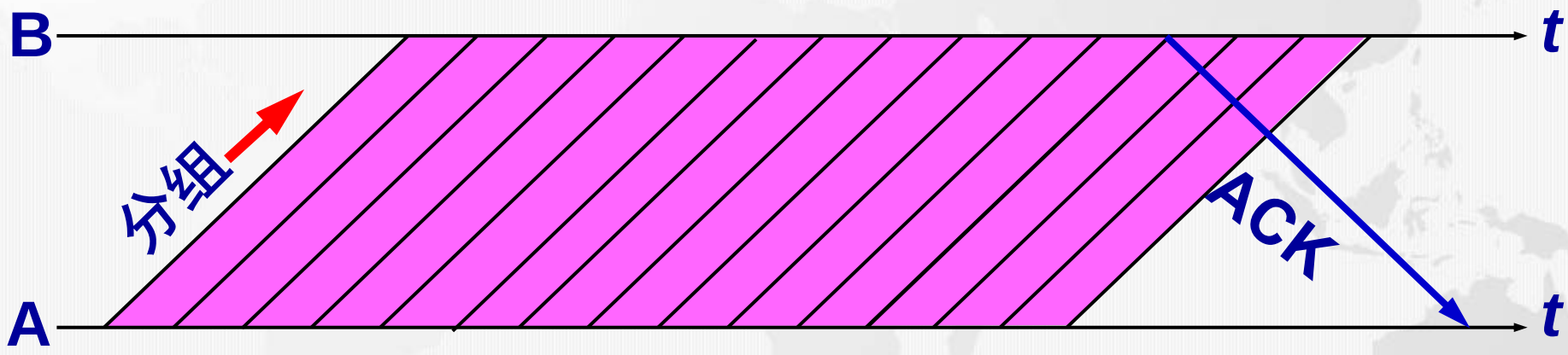


逐个确认



累积确认

☺ 对按序到达的最后一个分组发送确认，这样就表示：到这个分组为止的所有分组都已正确收到了



Go-back-N（回退 N）

- Go-back-N（回退 N），表示需要再退回来重传已发送过的 N 个分组
  - 如果发送方发送了前5个分组，而后面的第3个分组丢失了。这时接收方只能对前两个分组发出确认。发送方无法知道后面三个分组的下落，而只好把后面的三个分组都再重传一次

- 通信线路质量不好，连续ARQ协议会带来负面的影响

累积确认优缺点

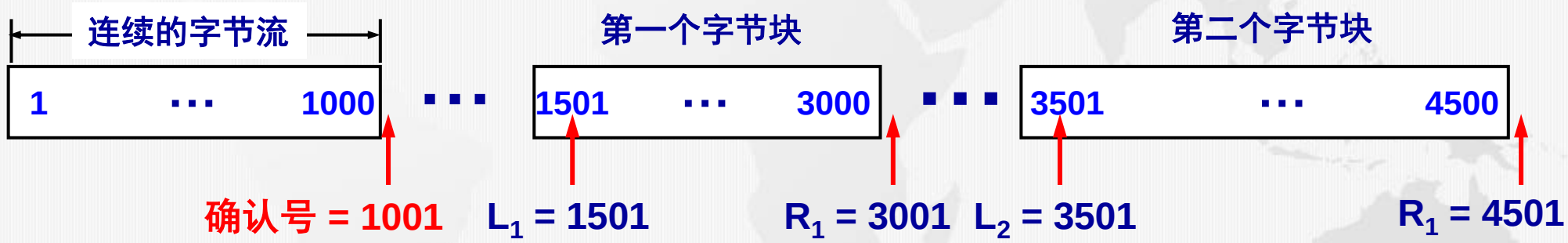
- ☺ 优点：容易实现，即使确认丢失也不必重传
- ☺ 缺点：不能向发送方反映出接收方已经正确收到的所有分组的信息

能否只传送缺少的数据而不重传已经正确到达接收方的数据

2.4 选择确认

接收到的字节流序号不连续

- ☺ TCP 的接收方在接收对方发送过来的数据字节流的序号不连续，结果就形成了一些不连续的字节块
  - ○ 每一个字节块都有两个边界：边界和右边界



选择确认SACK（Selective ACK)

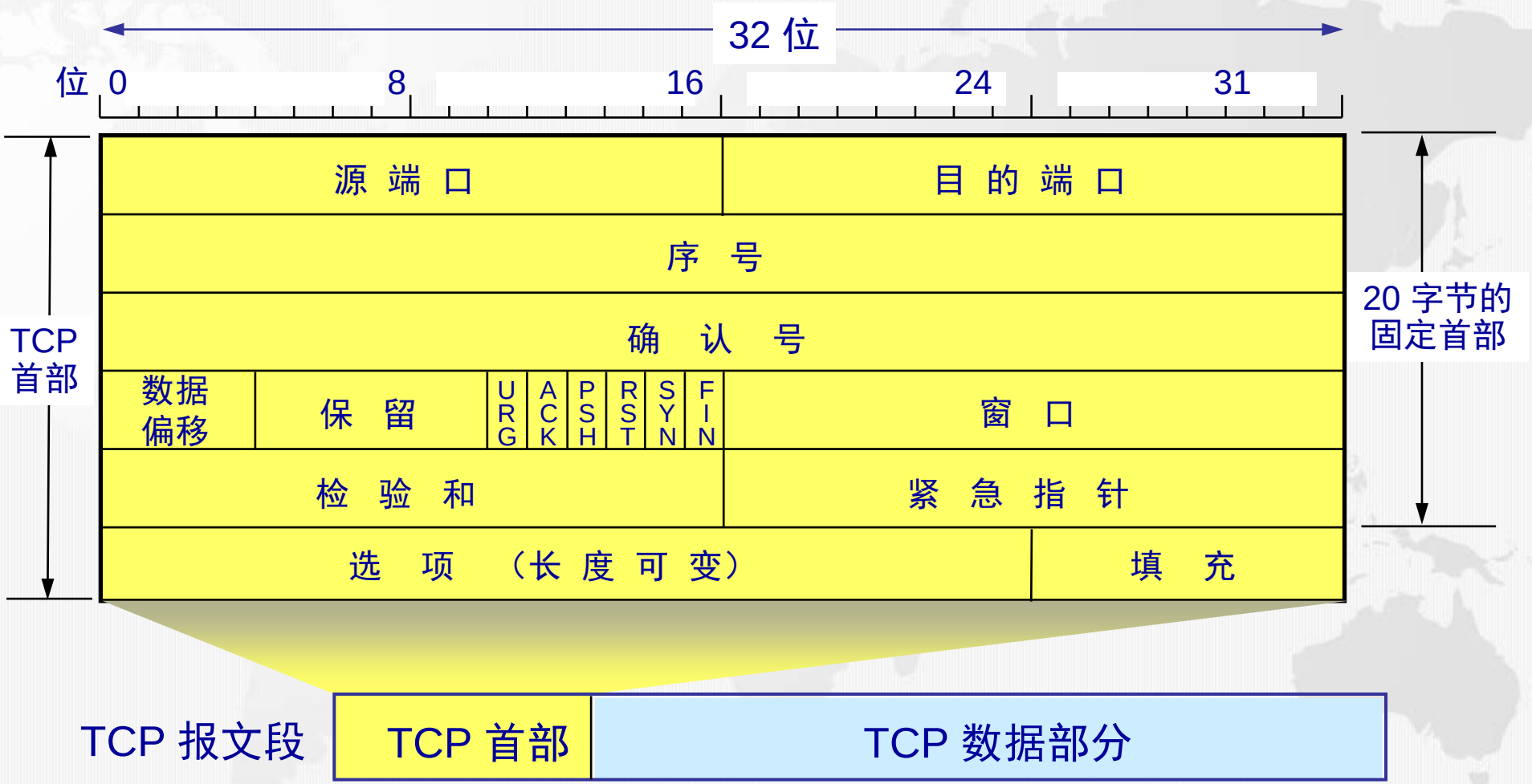
- ☺ 接收方收到了和前面的字节流不连续的两个字节块
  - 先收下这些数据
  - 把这些信息准确地告诉发送方，使发送方不要再重复发送这些已收到的数据

使用选择确认

- 使用选择确认，在建立TCP连接时，需在TCP首部的选项中加上“允许SACK”的选项，且双方必须都事先商定好

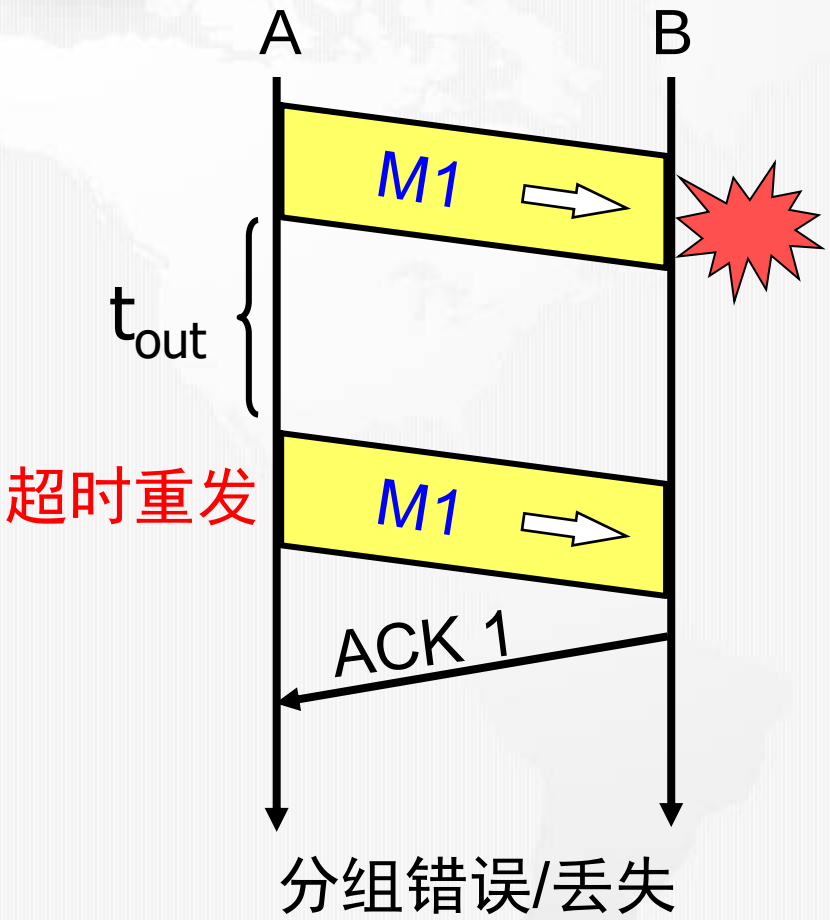
- 由于首部选项的长度最多只有40字节，而指明一个边界就要用掉4字节，因此在选项中最多只能指明4个字节块的边界信息。

TCP报文格式

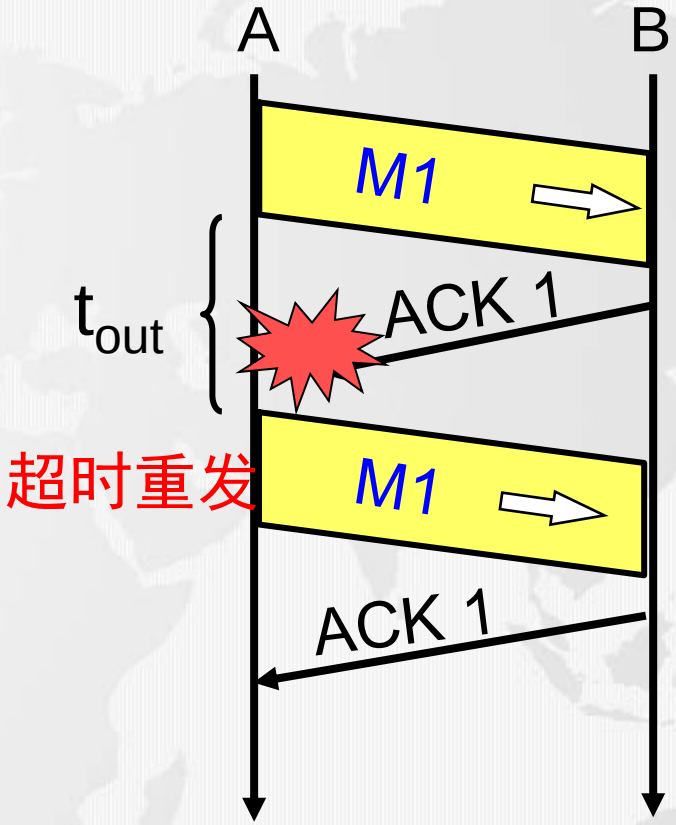

32 位
位 0
8
16
24
31
源 端 口
目 的 端 口
序 号
确 认 号
数据偏移
保 留
URG
ACK
PSH
RST
SYN
FIN
窗 口
检 验 和
紧 急 指 针
选 项 （长 度 可 变）
填 充
TCP 首部
20 字节的固定首部
TCP 报文段
TCP 首部
TCP 数据部分

2.5 超时计时器

出现差错，超时重传



分组错误/丢失



确认出错/丢失

超时计时器如何设置?

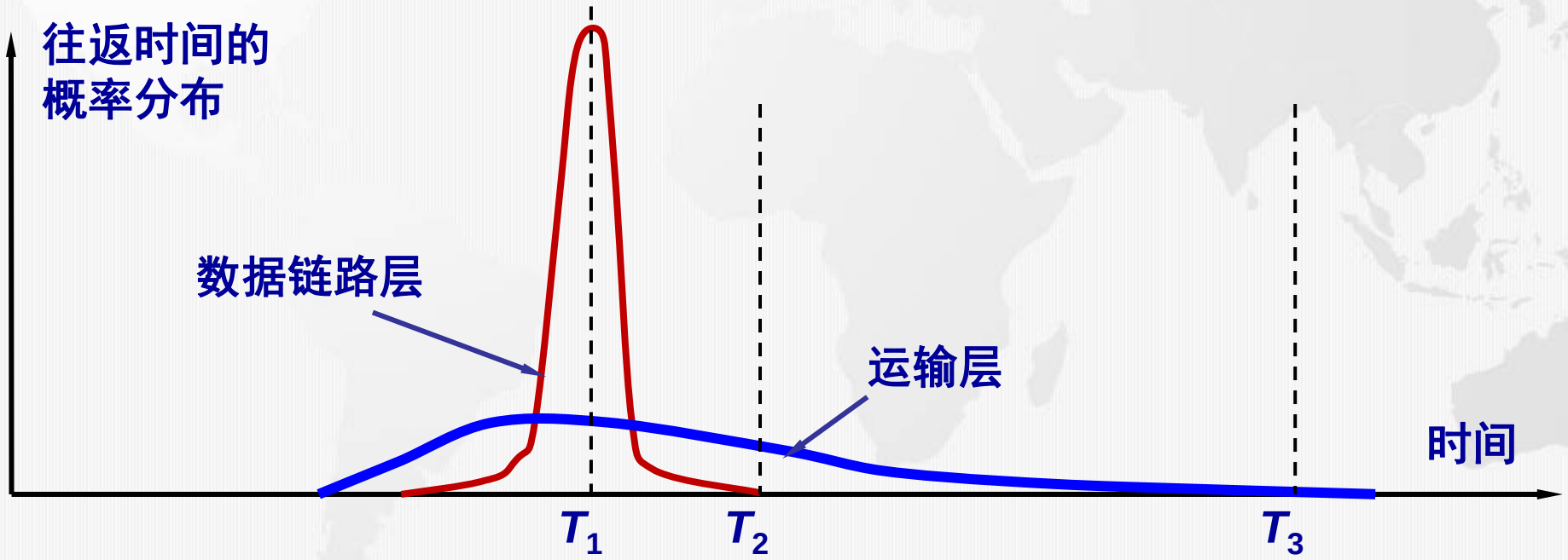
超时计时器的重传时间

- 超时重传时间太长、太短
  - 如果把超时重传时间设置得太短，就会引起很多报文段的不必要的重传，使网络负荷增大。
  - 但若把超时重传时间设置得过长，则又使网络的空闲时间增大，降低了传输效率

- 应当比数据在分组传输的平均往返时间更长一些

往返时延的方差很大

☺ TCP 的下层是一个互联网环境，IP 数据报所选择的路由变化很大。因而运输层的往返时间 (RTT) 的方差也很大



自适应算法

- 记录一个报文段发出的时间，以及收到相应的确认的时间。这两个时间之差就是报文段的往返时间RTT

加权平均往返时间

- TCP保留了RTT的一个加权平均往返时间RTT_S（这又称为平滑的往返时间）。
- 第一次测量到 RTT 样本时，RTT_S 值就取为所测量到的 RTT 样本值。以后每测量到一个新的 RTT 样本，就按下式重新计算一次 RTT_S：

$$\text{新的}RTT_S = (1-\alpha) \times (\text{旧的}RTT_S) + \alpha \times (\text{新的RTT样本})$$

- 其中，$0 \le \alpha < 1$。若α很接近于零，表示RTT值更新较慢。若选择α接近于 1，则表示 RTT 值更新较快。
- RFC 2988 推荐的α值为1/8，即0.125。

超时重传时间 RTO

- RTO (Retransmission Time-Out) 应略大于上面得出的加权平均往返时间 RTT_S。
- RFC 2988 建议使用下式计算 RTO：

$$RTO = RTT_S + 4 \times RTT_D$$

  - RTT_D 是 RTT的偏差的加权平均值
  - RTT_S是 RTT的加权平均往返时间
- RFC 2988 建议这样计算 RTT_D。第一次测量时，RTT_D 值取为测量到的 RTT 样本值的一半。在以后的测量中，则使用下式计算加权平均的 RTT_D：

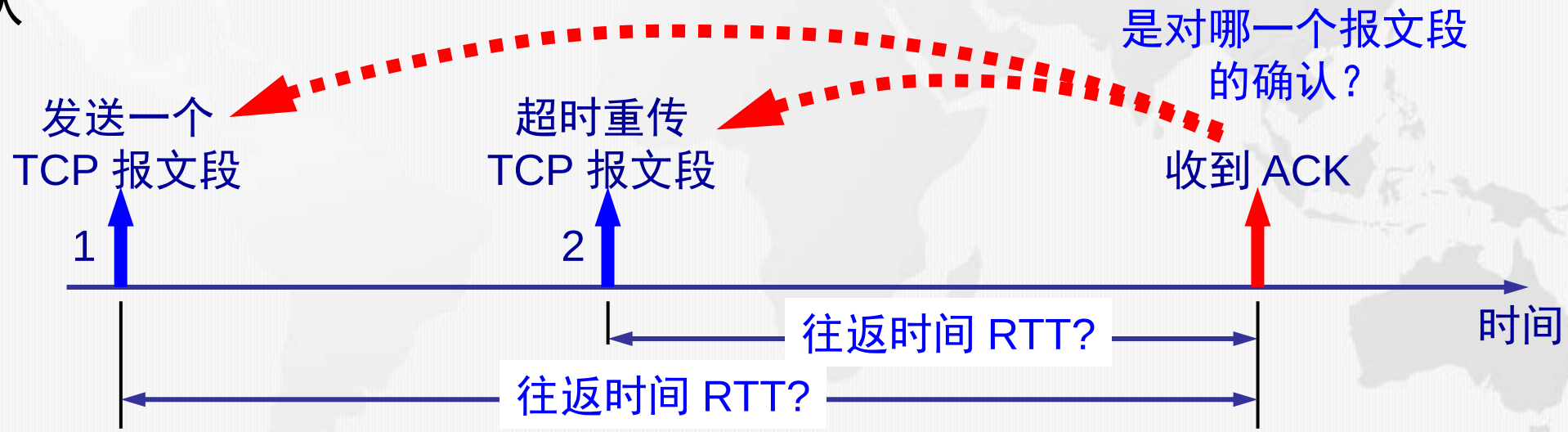
$$\text{新的 } RTT_D = (1 - \beta) \times (\text{旧的} RTT_D) + \beta \times |RTT_S - \text{新的 RTT 样本}|$$

- β是个小于1的系数，其推荐值是1/4，即 0.25

重传情况下，RTT怎么算?

往返时间 (RTT) 的测量相当复杂

- ☺ TCP 报文段1没有收到确认。重传（即报文段 2）后，收到了确认报文段 ACK
  - ○ 如何判定此确认报文段是对原来的报文段 1 的确认，还是对重传的报文段 2 的确认



(2) Karn算法

- ☺ 在计算平均往返时间RTT时，只要报文段重传了，就不采用其往返时间样本
  - 这样得出的加权平均平均往返时间RTT_S 和超时重传时间RTO较准确

这样处理，有缺陷吗？

Karn存在问题

- ☺ 重传，超时重传时间就无法更新
  - 当报文段的时延突然增大了很多时，在原来得出的重传时间内，不会收到确认报文段，于是重传报文段。但根据Karn算法，不考虑重传的报文段的往返时间样本。这样，一直处于重传状态。

(3) 修正的 Karn 算法

- ☺ 报文段每重传一次，就把 RTO 增大一些：

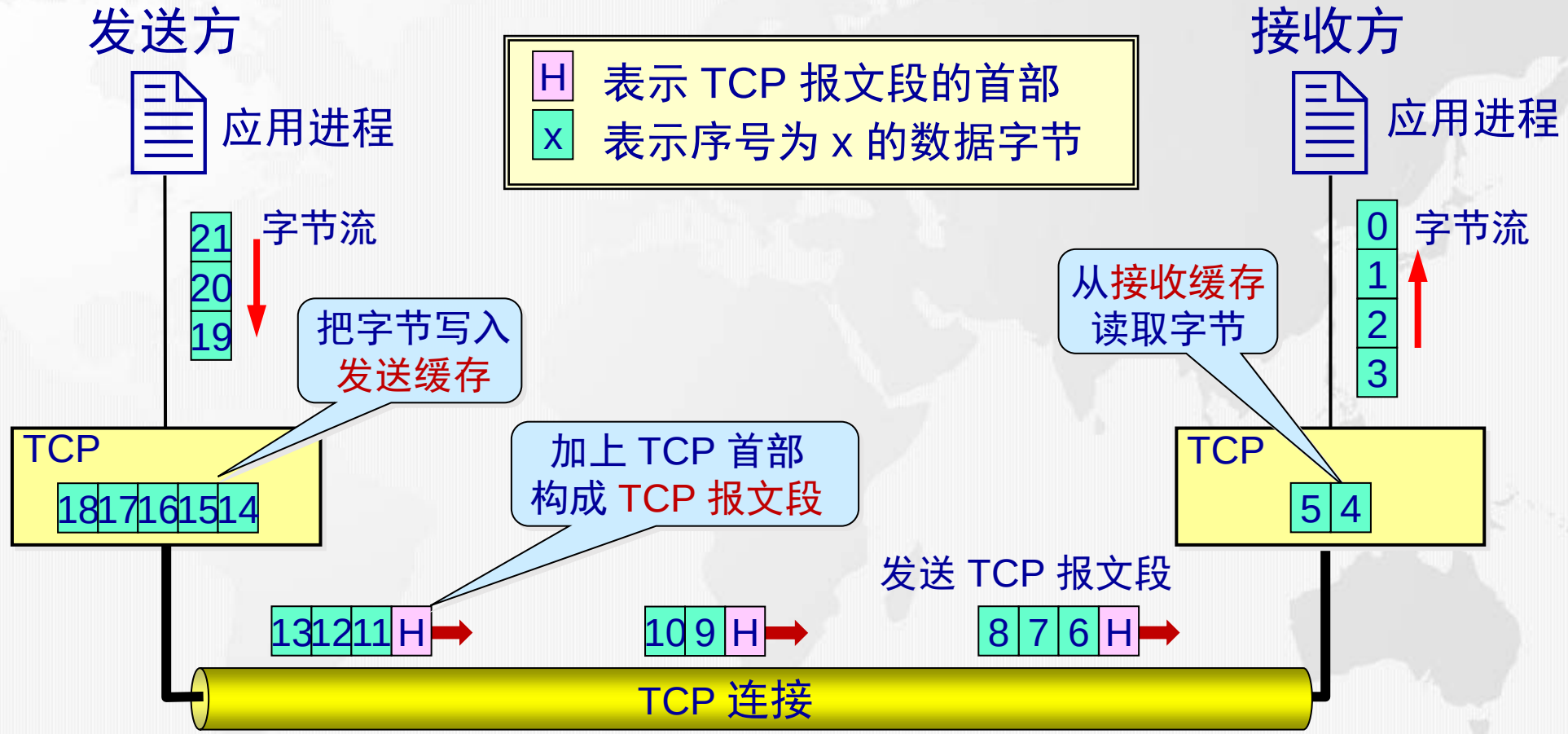
  新的 RTO = γ × (旧的 RTO)

  - ○ 系数 γ 的典型值是 2 。

- ☺ 当不再发生报文段的重传时，才根据报文段的往返时延更新平均往返时延 RTT 和超时重传时间RTO的数值
- ☺ 实践证明，这种策略较为合理

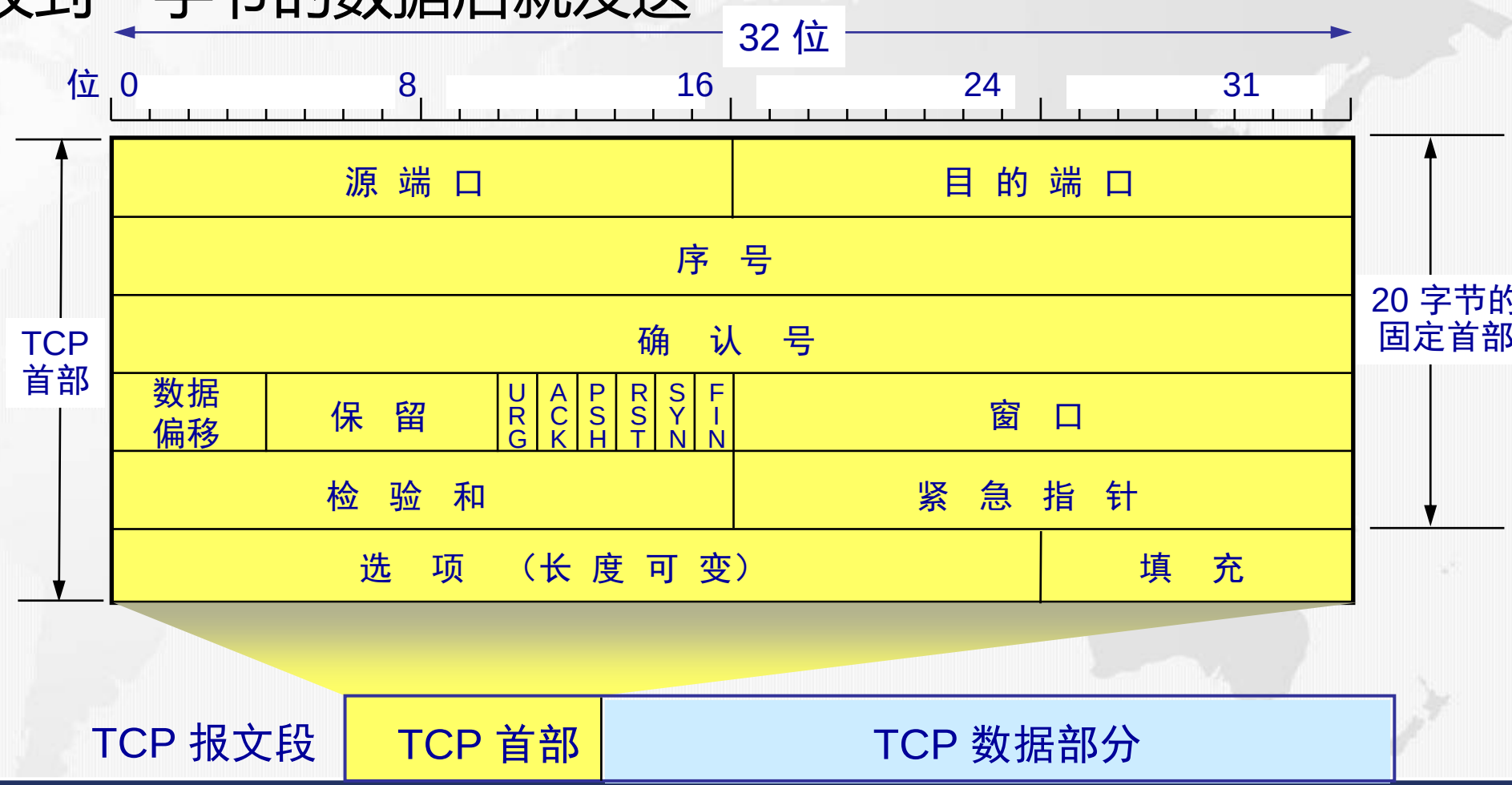
2.6 发送时机

什么时候把数据发出去?


发送方
应用进程
H 表示 TCP 报文段的首部
x 表示序号为 x 的数据字节
接收方
应用进程
21
20
19
字节流
把字节写入
发送缓存
从接收缓存
读取字节
0
1
2
3
字节流
TCP
18 17 16 15 14
加上 TCP 首部
构成 TCP 报文段
TCP
5 4
发送 TCP 报文段
13 12 11 H
10 9 H
8 7 6 H
TCP 连接

一有数据就发送

☺ 发送方每次接收到一字节的数据后就发送



(1) 达到某个值就发

☺ 维持一个变量，等于最大报文段长度MSS。只要缓存中存放的数据达到MSS字节时，就组装成一个TCP报文段发送出去

(2) 抢着操作

- ☺ 发送方的应用进程指明要求发送报文段，即TCP支持的推送 (push) 操作

(3) 超出计时器，发送

☺ 发送方的一个计时器期到了，这时就把当前已有的缓存数据装入报文段（但长度不能超过MSS）发送出去

Nagle算法

- 若发送方把要发送的数据逐个字节地送到TCP的发送缓存
  - 发送方就把第一个数据字节先发送出去，把后面到达的数据字节都缓存起来
  - 当发送方收到对第一个数据字符的确认后，再把发送缓存中的所有数据组装成一个报文段发送出去，同时继续对随后到达的数据进行缓存。
  - 只有在收到对前一个报文段的确认后才继续发送下一个报文段。
- 当到达的数据已达到发送窗口大小的一半或已达到报文段的最大长度时，就立即发送一个报文段。

目录
CONTENTS

理想传输

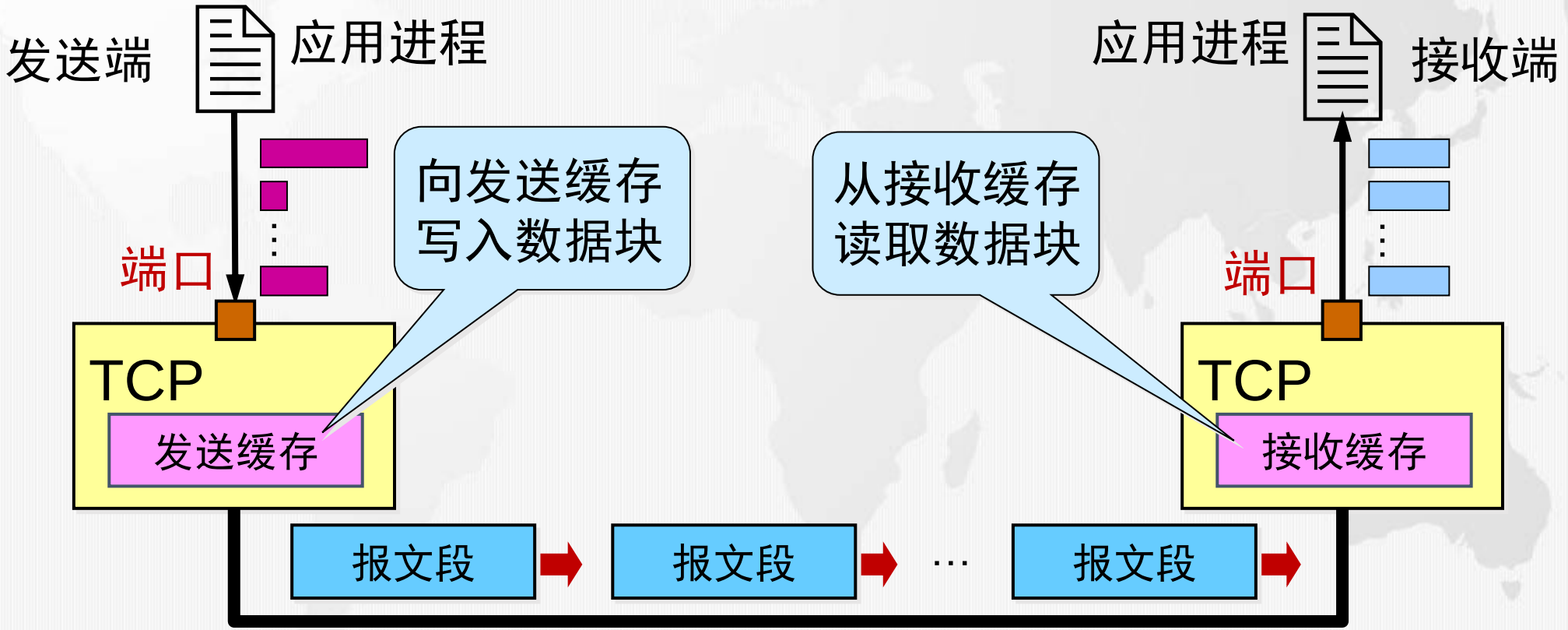
- 理想的传输条件有以下两个特点：
  - 传输信道不产生差错。
  - 不管发送方以多快的速度发送数据，接收方总是来得及处理收到的数据

- 然而实际的网络都不具备以上两个理想条件。必须使用一些可靠传输协议，在不可靠的传输信道实现可靠传输

怎么做到速度匹配?

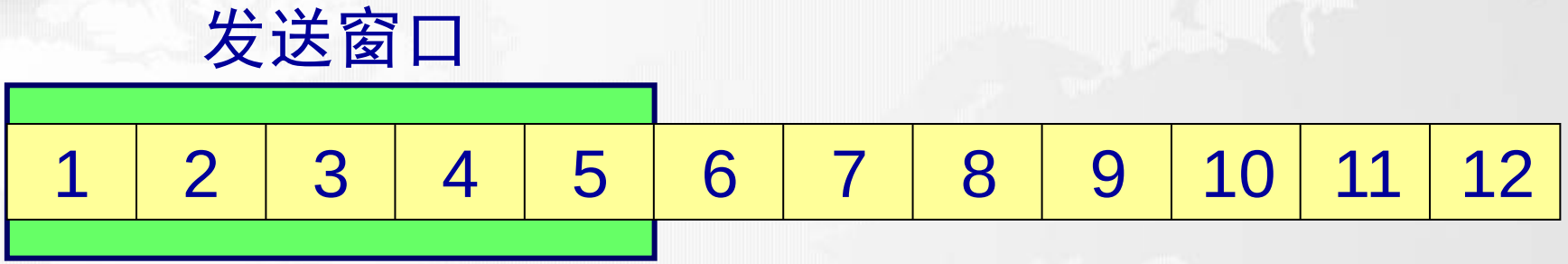
对字节流进行分段

☺ TCP不关心应用进程一次把多长的报文发送到TCP缓存

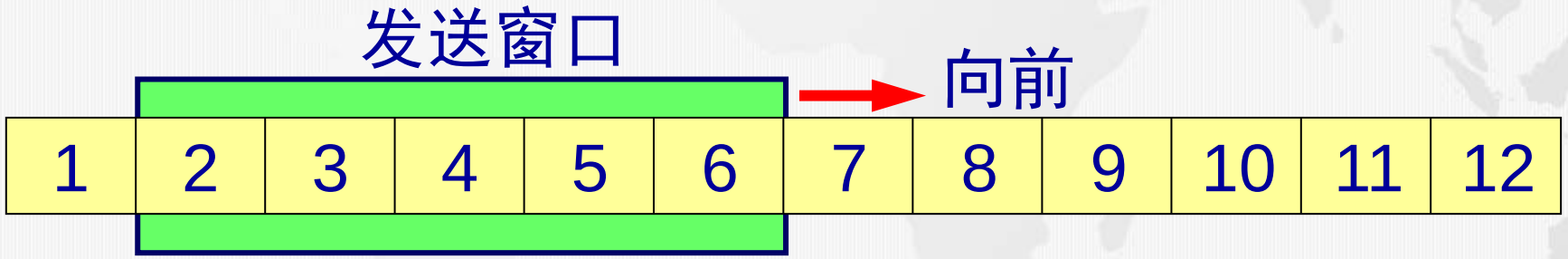


思路

连续ARQ工作原理



(a) 发送方维持发送窗口（发送窗口是5）



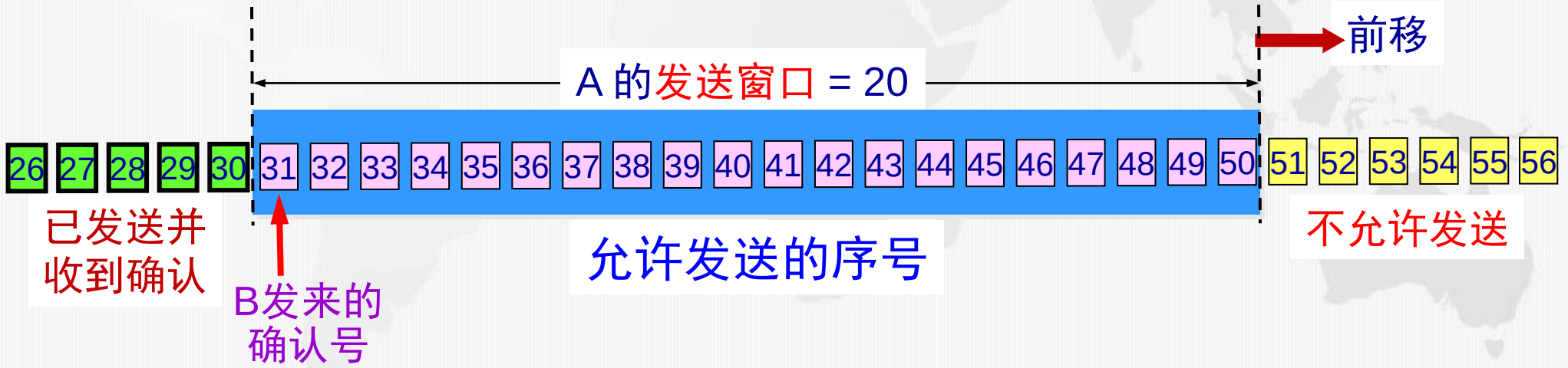
(b) 收到一个确认后发送窗口向前滑动

连续ARQ协议

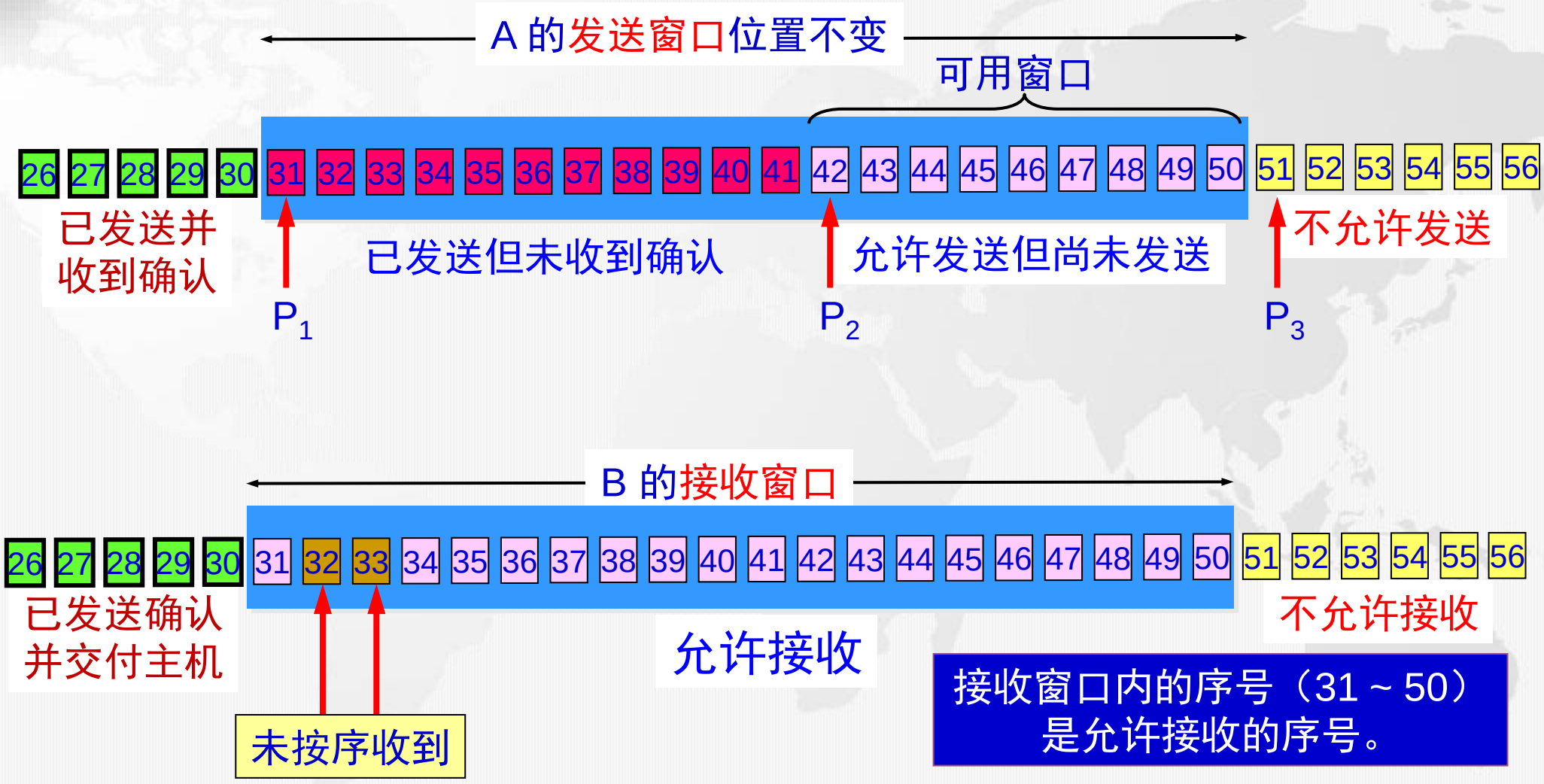
- 发送方维持的发送窗口
  - 位于发送窗口内的分组都可连续发送出去，而不需要等待对方的确认。信道利用率就提高了
  - 发送方每收到一个确认，就把发送窗口向前滑动一个位置

TCP的滑动窗口是以字节为单位

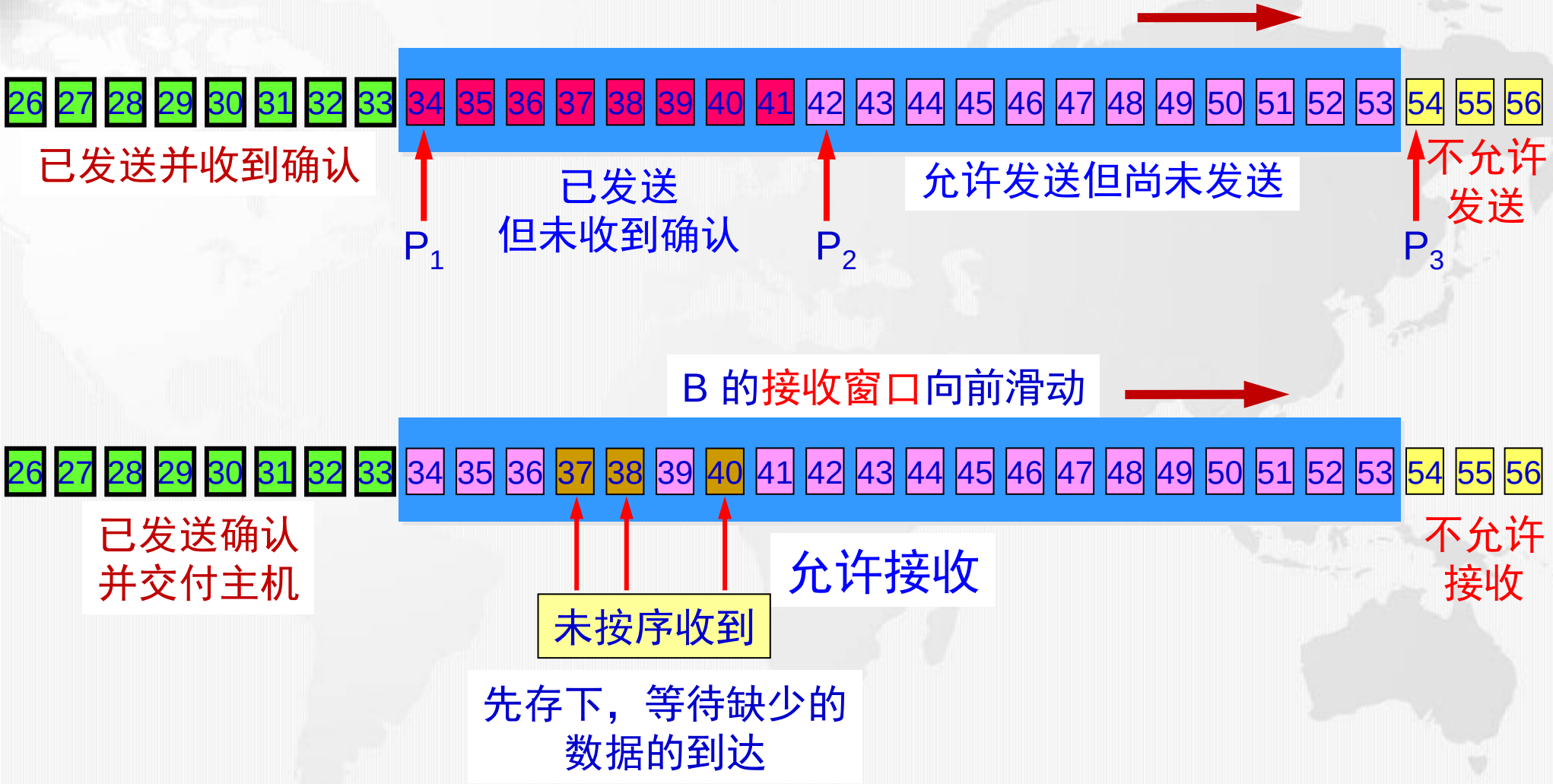
- B给出的窗口值，A构造出自己的发送窗口
  - A收到了B发来的确认报文段，窗口是20字节，而确认号是31
  - 确认号：31之前（不包含）的数据都已经收到了
  - 窗口：接收窗口，从本报文段确认号算起，接收方允许对方发送的数据量



A发送了11个字节的数据



A收到新的确认号34，发送窗口向前滑动

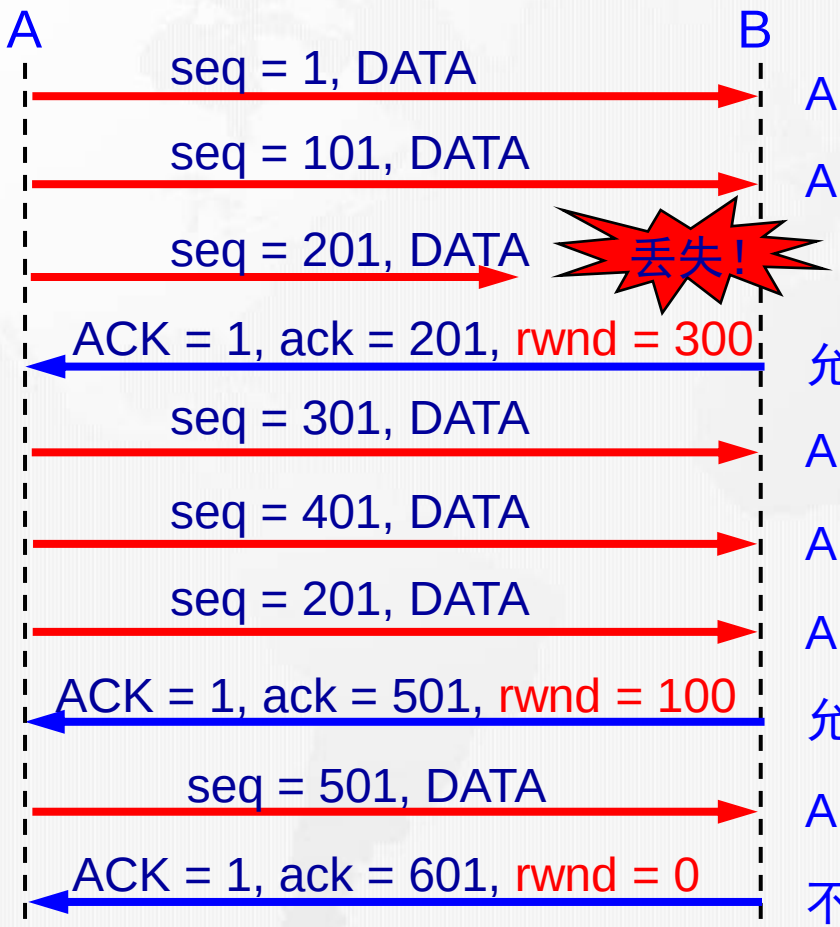


A的有效窗口为0


A 的发送窗口已满，有效窗口为零
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56
已发送并收到确认
已发送但未收到确认
不允许发送
P1
P2
P3

利用可变窗口进行流量控制

☺ A向B发送数据。B告诉A：我的接收窗口 rwnd = 400（字节）



A 发送了序号 1 至 100，还能发送 300 字节

A 发送了序号 101 至 200，还能发送 200 字节

允许 A 发送序号 201 至 500 共 300 字节

A 发送了序号 301 至 400，还能再发送 100 字节新数据

A 发送了序号 401 至 500，不能再发送新数据了

A 超时重传旧的数据，但不能发送新的数据

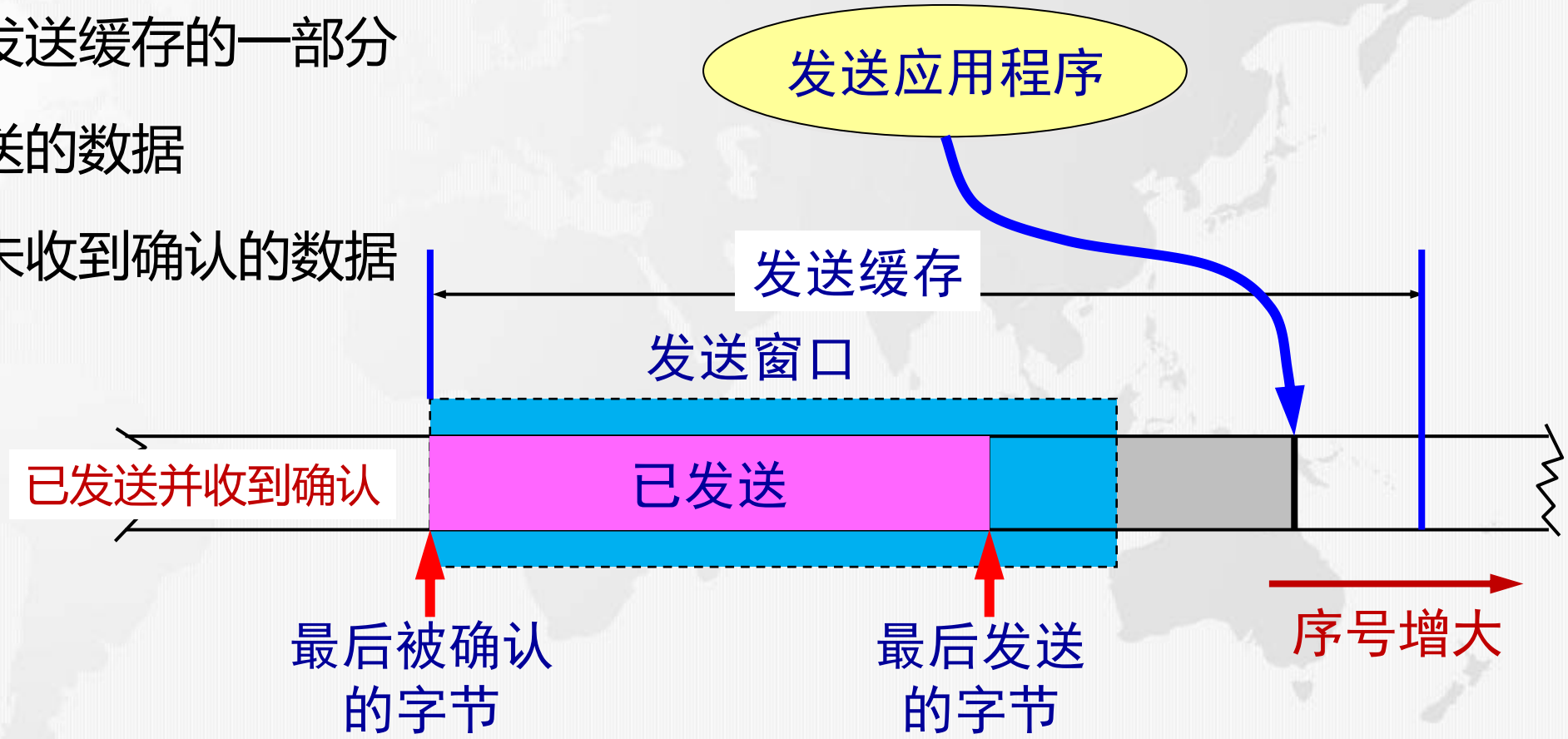
允许 A 发送序号 501 至 600 共 100 字节

A 发送了序号 501 至 600，不能再发送了

不允许 A 再发送（到序号 600 为止的数据都收到了）

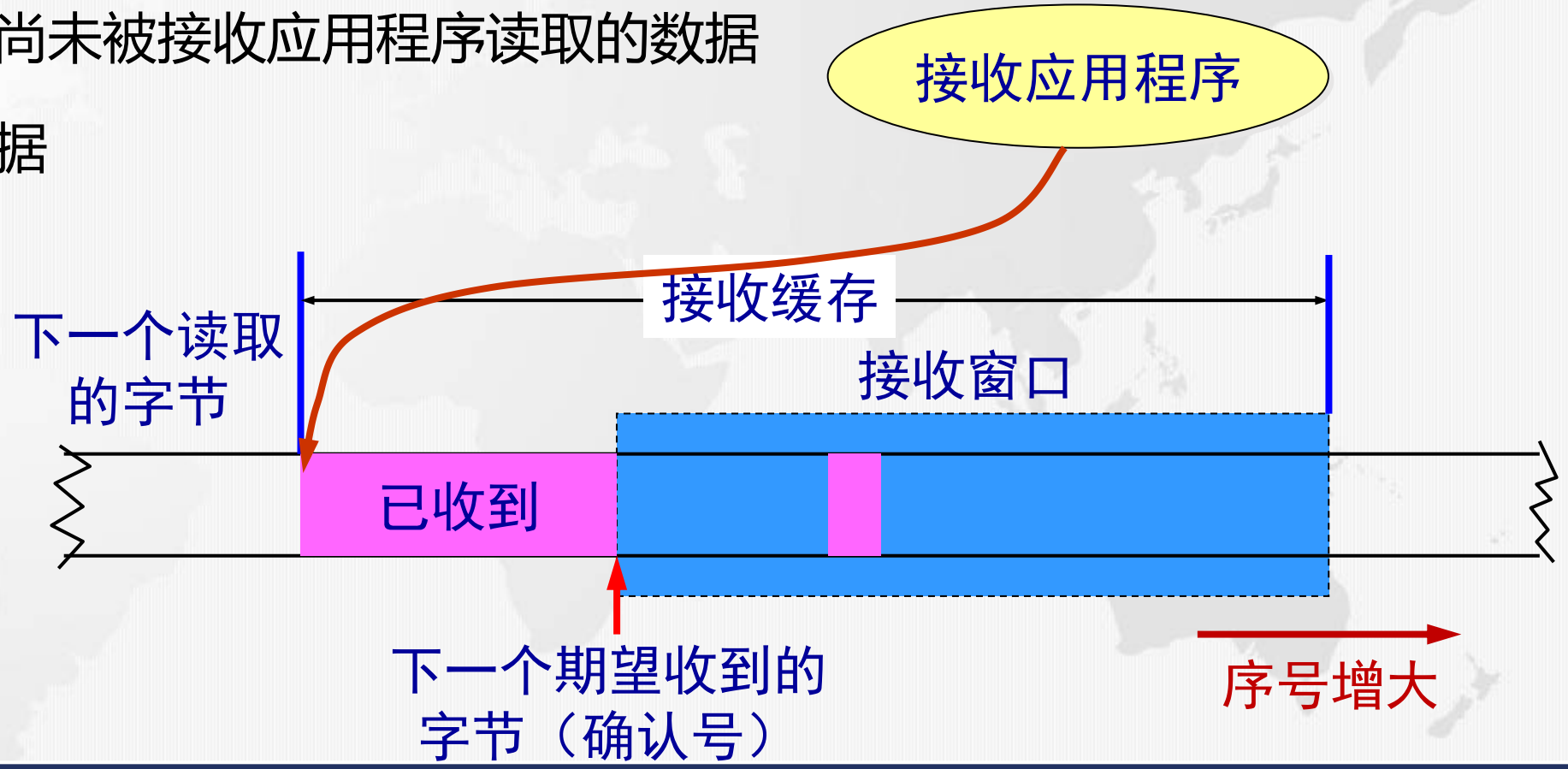
发送缓存与窗口的关系

- 发送方的应用进程把字节流写入TCP的发送缓存
  - 发送窗口只是发送缓存的一部分
  - 发送方准备发送的数据
  - 已发送出但尚未收到确认的数据



接收缓存与接收窗口

- 接收缓存用来暂时存放
  - 按序到达的、但尚未被接收应用程序读取的数据
  - 不按序到达的数据



思考

- B向A发送了零窗口的报文段后不久，B的接收缓存又有了一些存储空间
  - 于是B向A发送了rwnd = 400的报文段，但这个报文段在传送过程中丢失了
  - A一直等待收到B发送的非零窗口的通知，而B也一直等待A发送的数据
- 重传
- 其他解决死锁方法

持续计时器 (persistence timer)

- TCP为每一个连接设有一个持续计时器 (persistence timer)
  - 只要 TCP 连接的一方收到对方的零窗口通知，就启动该持续计时器
  - 若持续计时器设置的时间到期，就发送一个零窗口探测报文段（仅携带1字节的数据），而对方就在确认这个探测报文段时给出了现在的窗口值
    - 若窗口仍然是零，则收到这个报文段的一方就重新设置持续计时器
    - 若窗口不是零，则死锁的僵局就可以打破了

几个注意点

- A的发送窗口并不总是和B的接收窗口一样大（更新滞后）
- TCP标准没有规定对不按序到达的数据应如何处理。通常是先临时存放在接收窗口中，等到字节流中所缺少的字节收到后，再按序交付上层的应用进程
- TCP要求接收方必须有累积确认的功能，减小传输开销
- TCP是全双工通信，每一方都有发送窗口和接收窗口

接收方发送确认

- ☺ 接收方可以在合适的时候发送确认，也可以在自己有数据要发送时把确认信息顺便捎带上
  - 接收方不应过分推迟发送确认，否则会导致发送方不必要的重传，反而浪费了网络的资源

Q & A

qiankun.su@jmu.edu.cn

苏铅坤