**20181102**

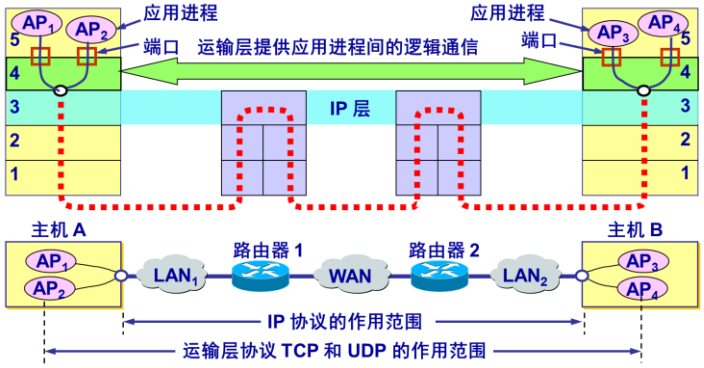
# 第 5 章　运输层

## 🕮　5.1　运输层协议概述

### ❀　5.1.1　进程之间的通信

从通信和信息处理的角度看，运输层向上面的应用层提供通信服务，它属于面向通信部分的最高层，同时也是用户功能中的最低层。
当网络的边缘部分中的两个主机使用网络的核心部分的功能进行端到端的通信时，只有主机的协议栈才有运输层，而网络核心部分中的路由器在转发分组时都只用到下三层的功能。

如图是运输层作用的示意图：




从 IP 层来说，通信的两端是两台主机。但严格地讲，两台主机进行通信是指两台主机中的**应用进程互相通信**。
虽然 IP 协议实现将分组送到目的主机，但此分组仍停留在网络层，而没有交付主机的应用进程。从运输层角度看，通信的真正端点并不是主机而是**主机中的进程**。也就是说，端到端的通信是应用进程之间的通信。

在一台主机中经常有多个应用进程同时分别和另一台主机的多个应用进程通信。如上图的主机 A 和 B 的$AP_1 \leftrightarrow AP_3$和$AP_2 \leftrightarrow AP_4$。这表明运输层有一个很重要的功能：复用(multiplexing)和分用(demultiplexing)。
"复用"是指发送方不同应用进程都可以使用同一个运输层协议传送数据；
"分用"是指接收方的运输层在剥去报文首部后能正确交付目的应用进程。

"运输层提供应用进程间的逻辑通信"是指运输层向高层用户屏蔽了下面网络核心的细节(如网络拓扑、所采用的路由选择协议等)，使应用进程看起来好像在两个运输层之间直接传送数据，而实际上并没有水平方向的物理连接。

根据应用程序的不同需求，运输层需要有两种不同的运输协议：面向连接的 TCP 和无连接的 UDP。
当采用面向连接的 TCP 协议时，尽管下面的网络是不可靠的，但这种逻辑通信就相当于一条全双工的可靠信道。
当采用无连接的 UDP 协议时，这种逻辑通信信道仍是不可靠的。

网络层和运输层的区别：网络层为主机之间提供逻辑通信，而运输层为应用进程之间提供端到端的逻辑通信。


### ❀ 5.1.2 运输层的两个主要协议

TCP/IP 运输层两个主要协议：

(1) 用户数据报协议 UDP (User Datagram Protocol)

(2) 传输控制协议 TCP (Transmission Control Protocol)

如图是两者在协议栈中的位置：


OSI 中，两个对等运输实体在通信时传送的数据单位叫作运输协议数据单元 TPDU (Transport Protocol Data Unit)。

TCP/IP 体系中，TCP 数据单元是 TCP 报文段(segment)；UDP 数据单元是 UDP 用户数据报。

UDP：在传送数据之前不需要先建立连接。对方的运输层收到 UDP 报文后，不需要给出任何确认。虽然 UDP 不提供可靠交付，但某些情况 UDP 是一种最有效的工作方式。

TCP：提供面向连接的服务。传送数据前必须先建立连接，传送结束要释放连接。不提供广播或多播服务。由于提供可靠的、面向连接的运输服务，因此不可避免地增加了许多的开销(如确认、流量控制、计时器、连接管理等)。这不仅使协议数据单元的首部增大很多，还要占用许多的处理机资源。

### ❀ 5.1.3 运输层的端口

计算机中的进程是用进程标识符(Process Identifier)来标志的，它是由操作系统所指派的。但不能用它来标志运行在应用层的各种应用进程，因为在互联网上使用的计算机操作系统种类很多，而不同的操作系统又使用不同格式的进程标识符。因此，必须使用统一的方法对 TCP/IP 体系的应用进程进行标志。

使用 `tasklist` 查看所有进程及对应 PID：(或：任务管理器->服务)

```
C:\Users\hikari>tasklist

映像名称                       PID 会话名              会话#       内存使用
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0         24 K
```

```
System                          4 Services                  0        388 K
smss.exe                      416 Services                  0      1,300 K
csrss.exe                     652 Services                  0      6,072 K
wininit.exe                   740 Services                  0      5,684 K
csrss.exe                     760 Console                   1     95,028 K
services.exe                  824 Services                  0     10,640 K
……
```

不能把特定进程作为通信的最后终点，因为进程的创建和撤销都是动态的，发送方几乎无法识别对方机器上的进程。另外，往往需要利用目的主机提供的功能来识别终点，而不需要知道具体实现此功能的进程是哪一个。

解决方法是在运输层使用协议端口号(protocol port number)，简称端口(port)。
虽然通信的终点是应用进程，但只要将传送的报文交给目的主机某个合适的目的端口，剩下的工作(最后交付目的进程)就由 TCP 或 UDP 完成。

**注意**：在协议栈层间的抽象的协议端口是软件端口；和路由器或交换机上的硬件端口是完全不同的概念。硬件端口是不同硬件设备进行交互的接口，而软件端口是应用层的各种协议进程与运输实体进行层间交互的一种地址。

运输层用一个 16 位端口号标志一个端口。端口号只具有本地意义，只是为了标志本计算机应用层中的各进程在和运输层交互时的层间接口。不同计算机的相同端口号是**没有联系**的。

由此可见，两个计算机中的进程要互相通信，不仅必须知道对方的 IP 地址(为了找到目的主机)，而且要知道对方的端口号(为了找到目的应用进程)。

互联网计算机通信采用 C/S 方式。客户端在发起通信请求时，必须先知道服务器 IP 地址和端口号。因此运输层端口号分为两类：

**① 服务器端使用的端口号**

1) 熟知端口号(well-known port number)或系统端口号，数值为 0~1023。
IANA 将其指派给 TCP/IP 最重要一些应用程序，让所有用户都知道。

| 应用程序 | FTP | TELNET | SMTP | DNS | TFTP | HTTP | SNMP | SNMP(trap) | HTTPS |
|---|---|---|---|---|---|---|---|---|---|
| 熟知端口号 | 21 | 23 | 25 | 53 | 69 | 80 | 161 | 162 | 443 |

2) 登记端口号(registered port number)或已注册端口号，数值为 1024~49151。
为没有熟知端口号的应用程序使用的。

**② 客户端使用的端口号**

也称短暂端口号或动态端口号或私有端口号(ephemeral / dynamic / private port number)，数值为 49152~65535，留给客户进程选择暂时使用。
当服务器进程收到客户进程的报文时，就知道客户进程所使用的动态端口号，因而可以把数据发送给客户进程。通信结束后，刚才使用的客户端口号不复存在，这个端口号就可供其他客户进程使用。

**20181104**

## 🕮 5.2 用户数据报协议 UDP

### ❁ 5.2.1 UDP 概述

UDP 只在 IP 的数据报服务之上增加了很少一点的功能：

1) 复用和分用的功能
2) 差错检测的功能

UDP 的主要特点：

(1) UDP 是无连接的，发送数据之前不需要建立连接，因此减少了开销和发送数据之前的时延。

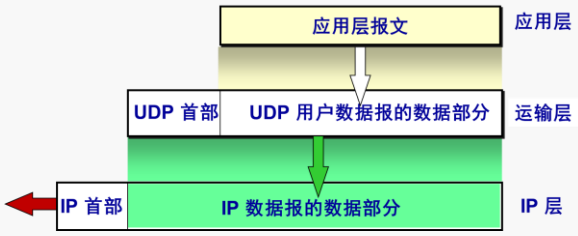
(2) UDP 使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的连接状态表。

(3) UDP 是面向报文的。

UDP 对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界，在添加首部后就向下交付 IP 层。也就是应用层交给 UDP 多长的报文，UDP 就照样发送，即一次发送一个报文。

接收方 UDP 对 IP 层交上来的 UDP 用户数据报，在去除首部后就原封不动地交付上层的应用进程，一次交付一个完整的报文。

应用层必须选择合适大小的报文：若报文太长，IP 层在传送时可能要进行分片；若太短，会使 IP 数据报的首部的相对长度太大，都会降低 IP 层效率。




(4) UDP 没有拥塞控制，因此网络出现的拥塞不会使源主机的发送速率降低。这对某些实时应用(如 IP 电话、实时视频会议等)是很重要的。

(5) UDP 支持一对一、一对多、多对一和多对多的交互通信。

(6) UDP 的首部开销小，只有 8 个字节，比 TCP 的 20 个字节的首部要短。

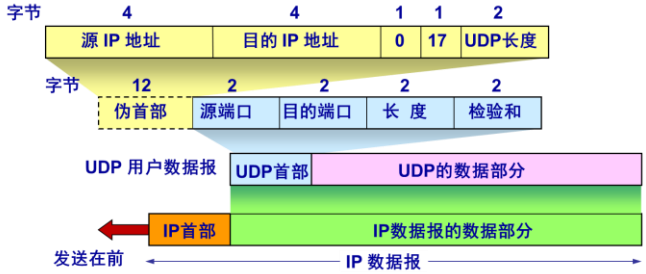
当很多源主机同时向网络发送高速率的实时视频流时，网络可能发生拥塞，结果都无法正常接收。因此不使用拥塞控制功能的 UDP 有可能会引起网络严重的拥塞问题。

### ❁ 5.2.2 UDP 的首部格式

用户数据报 UDP 有两个字段：数据字段和首部字段。首部字段很简单，只有 8 个字节，由 4 个字段组成，每个字段都是 2 字节。

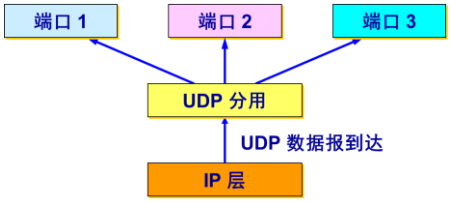
(1) 源端口：在需要对方回信时选用，不需要可用全 0
(2) 目的端口：交付报文时必须使用
(3) 长度：UDP 用户数据报的长度，最小值是 8 (只有首部)
(4) 检验和：检测 UDP 用户数据报在传输中是否有错，有错就丢弃

UDP 用户数据报的首部：




当运输层从 IP 层收到 UDP 数据报时，就根据首部的目的端口，把 UDP 数据报通过相应的端口，上交最后的终点，即应用进程。

如图是 UDP 基于端口分用的示意图：

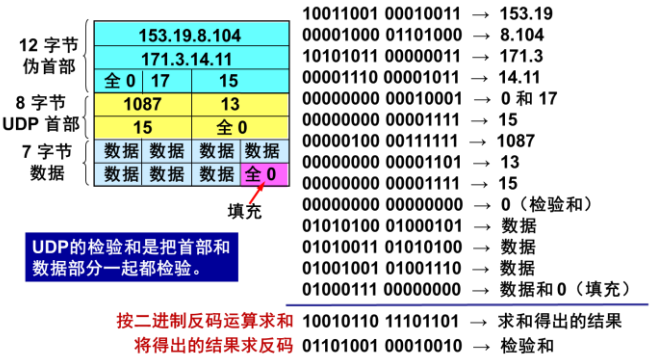



如果接收方 UDP 发现收到的报文目的端口号不存在对应的应用进程，丢弃该报文，由 ICMP 发送"端口不可达"差错报文给发送方。

**注意**：虽然在 UDP 之间的通信要用到其端口号，但由于 UDP 的通信是无连接的，因此不需要使用套接字 (TCP 通信必须在两个套接字间建立连接)。

UDP 计算检验和方法有点特殊，要在 UDO 用户数据报前增加 12 自己的伪首部，这种伪首部不是 UDP 真正的首部，而是为了计算检验和，临时添加的。

IP 数据报的检验和只检验首部，但 UDP 的检验和是将**首部和数据部分**一起都检验。如图，假设用户数据报长度为 15 字节，需要填充 1 字节的全 0，因为计算检验和是每 16 位一起计算的。




伪首部的 17 是 IP 数据报首部的协议字段值，17 对应就是 UDP；伪首部的 15 和 UDP 首部的 15，是 UDP 用户数据报的长度。

使用 Python 和 NetAssis 模拟聊天(UDP)：

```
from socket import AF_INET, SOCK_DGRAM, socket
from threading import Thread
```

```python
class UDPTest():
    def __init__(self):
        self.udp = socket(AF_INET, SOCK_DGRAM)
        # 对方 ip+port,此处使用 NetAssist 模拟另一个人,但编码是 GBK
        self.other_addr = ('192.168.43.117', 8080)
        self.addr = ('', 12345)  # 本方 ip+port
        self.BUFSIZE = 1024

    def recieve(self):
        while True:
            recv_msg, recv_addr = self.udp.recvfrom(self.BUFSIZE)
            # 加上\r 可以将本行内容覆盖
            print('\r>>[{}:{}]: {}\n<<'.format(
                *recv_addr, recv_msg.decode('GBK')), end='')

    def send(self):
        while True:
            send_msg = input('<<')  # 将输入内容发送
            self.udp.sendto(send_msg.encode('GBK'), self.other_addr)

    def __call__(self):
        self.udp.bind(self.addr)  # 绑定本方端口号
        # 本方开启两个线程,既可以发送,也可以接收
        Thread(target=self.recieve).start()
        Thread(target=self.send).start()

if __name__ == '__main__':
    UDPTest()()
```

测试：

```
<<你好
>>[192.168.43.117:8080]: Nice to meet you!
<<不要说鸟语！
>>[192.168.43.117:8080]: さらば、吾が友よ！
<<...
```

网络调试助手 NetAssis：

```
【Receive from 192.168.43.117 : 12345】 :
【2018-11-04 22:35:20:384】你好
【2018-11-04 22:35:48:587】不要说鸟语！
【2018-11-04 22:36:04:950】...
```

## 🕮 5.3 传输控制协议 TCP 概述

### ❁ 5.3.1 TCP 最主要的特点

TCP 是 TCP/IP 体系中非常复杂的一个协议。主要特点：

(1) TCP 是面向连接的运输层协议。应用程序在使用 TCP 协议前，必须建立 TCP 连接；传送数据结束后，必须释放 TCP 连接，如同打电话一般。

(2) 每一条 TCP 连接只能有两个端点(endpoint)，每一条 TCP 连接只能是点对点的(一对一)。

(3) TCP 提供可靠交付的服务。数据无差错、不丢失、不重复、按序到达。

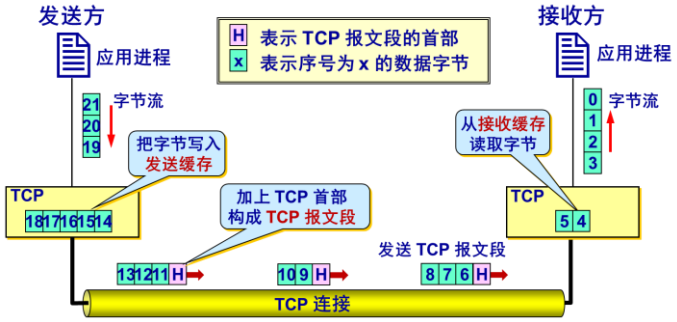
(4) TCP 提供全双工通信。允许双方应用进程都能发送数据。TCP 连接的两端都有发送缓存和接收缓存，用来临时存放双向通信的数据。

(5) 面向字节流。TCP 中的"流"(stream)是流入或流出进程的字节序列。

"面向字节流"的含义：虽然应用程序和 TCP 的交互是一次一个数据块(大小不等)，

但 TCP 把应用程序交下来的数据仅仅看成是一连串*无结构的字节流*。TCP 不知道传送的字节流的含义。

TCP 不保证接收方应用程序所收到的数据块和发送方发出的数据块具有相同数量、大小。但接收方应用程序收到的字节流必须和发出的字节流完全一样。
接收方应用程序必须有能力识别收到的字节流，并还原成有意义的数据。




**注意：**
1) 实际网络一个 TCP 报文段常常包含上千个字节，图中只画了几个字节。
2) TCP 连接是一条*虚连接*(逻辑连接)而不是一条真正的物理连接。
3) TCP 和 UDP 发送报文时采取的方式完全不同。TCP 不关心应用进程一次把多长的报文发送到 TCP 缓存，而是根据对方给出的*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字节。UDP 发送的报文长度是应用进程给出的。
4) 如果应用进程传到 TCP 缓存的数据块太大，TCP 就将其划分短一些再传送。
5) 如果应用进程一次只发来一个字节，TCP 也可等待积累有足够多字节后再构成报文段发送。

### 5.3.2 TCP 的连接

TCP 把*连接*作为最基本的抽象。
每一条 TCP 连接有两个端点，而 TCP 连接的端点叫*套接字*(socket)或*插口*。
端口号*拼接到*(concatenated with) IP 地址即构成套接字。

套接字 socket = (IP 地址: 端口号)

每一条 TCP 连接唯一地被通信两端的两个端点(即两个套接字)所确定。即：

TCP 连接 ::= {socket$_1$, socket$_2$} = {(IP$_1$: port$_1$), (IP$_2$: port$_2$)}

总之，TCP 连接是由协议软件所提供的一种抽象。TCP 连接的端点是个很抽象的套接字，即(IP 地址: 端口号)。同一个 IP 地址可以有多个不同的 TCP 连接，而且同一个端口号也可以出现在多个不同的 TCP 连接中。

**注意：**随着互联网的发展，socket 可表达多种意思：
(1) 允许应用程序访问连网协议的*应用编程接口* API (Application Programming Interface)，即运输层和应用层之间的一种接口，称为 socket API，简称 socket。
(2) socket API 中使用的一个*函数名*也叫 socket。
(3) 调用 socket 函数的*端点*称为 socket，如"创建一个数据报 socket"。
(4) 调用 socket 函数时其*返回值*称为 socket 描述符，可简称为 socket。
(5) 在操作系统内核中连网协议的 Berkeley 实现，称为 *socket 实现*。

**20181108**

## 🕮 5.4 可靠传输的工作原理

理想的传输条件的两个特点：

1) 传输信道不产生差错。
2) 不管发送方以多快的速度发送数据，接收方总来得及处理收到的数据。

实际的网络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。

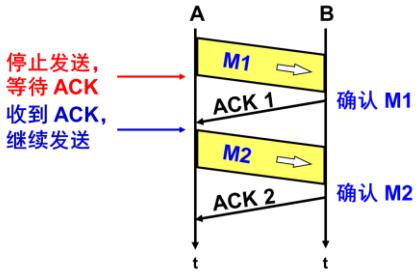
### ⊛ 5.4.1 停止等待协议

全双工通信的双方既是发送方也是接收方。为了方便，此处仅考虑 A 发送数据而 B 接收数据并发送确认；且传送的数据都称为分组，而不考虑数据在哪个层上传送。

"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。

#### ① 无差错情况

A 发送分组 M1，发完就暂停发送，等待 B 的确认(ACK)。B 收到 M1，就向 A 发送 ACK。A 在收到对 M1 的确认后，就再发送下一个分组 M2。



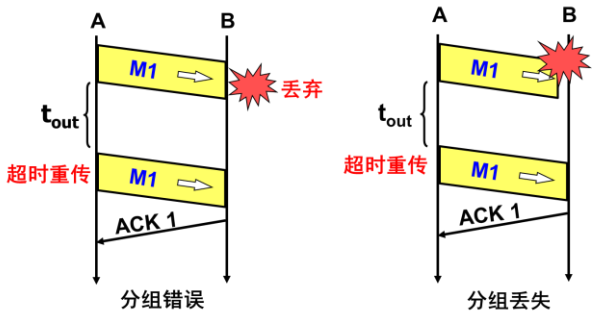

#### ② 出现差错

接收方 B 会出现两种情况：

1) B 接收 M1 时检测出了差错，就丢弃 M1，其他什么也不做(不通知 A)；
2) M1 在传输过程中丢失了，这时 B 什么都不知道，也什么都不做。

可靠传输协议使用：超时重传

A 为每个已发送的分组都设置了一个超时计时器。只要在超时计时器到期之前收到了对方的确认，就撤销该超时计时器，继续发送下一个分组；如果没有收到确认，就认为该分组丢失了，需要重传。




**注意：**

1) 发送完一个分组后，必须暂时保留已发送的分组的副本，以备重传；收到相

应确认后，可以删除分组副本。
2) 分组和确认分组都必须进行编号。这样才能知道收到了哪些分组的确认。
3) 超时计时器的重传时间应当比数据在分组传输的平均往返时间更长一些。

### ③ 确认丢失和确认迟到

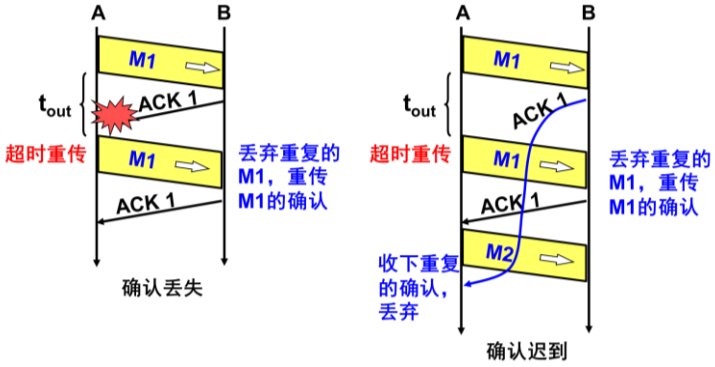
确认丢失

若 B 发送的对 M1 的确认丢失了，那么 A 在设定的超时重传时间内不能收到确认，就要重传 M1。假定 B 又收到了重传的分组 M1。这时 B 应采取：

1) **丢弃**此**重复**的分组，不向上层交付；
2) 向 A **发送确认**。不能认为已经发送过确认就不再发送，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。对重复的确认收下后就直接丢弃。B 仍会收到重复的 M1，并且同样要丢弃重复的 M1，并重传确认分组。

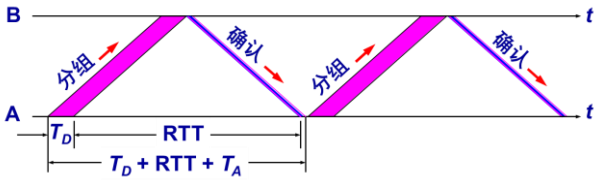



通常 A 最终总可以收到对所有发出分组的确认。如果 A 不断重传分组但总是收不到确认，就说明通信线路太差，不能进行通信。

使用上述的确认和重传机制，就可以在不可靠的传输网络上**实现可靠的通信**。
上述这种可靠传输协议常称为自动重传请求 ARQ (Automatic Repeat reQuest)。意思是重传的请求是自动进行的，接收方不需请求发送方重传某个出错的分组。

### ④ 信道利用率

停止等待协议的优点是简单，缺点是**信道利用率太低**。



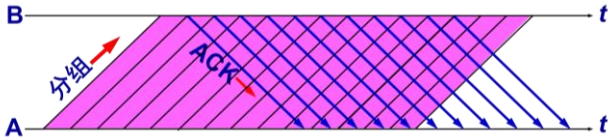

A 发送分组时间为$T_D$；B 发送确认分组时间$T_A$；RTT是往返时间。

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

可以看出，当往返时间RTT远大于分组发送时间$T_D$时，信道的利用率非常低。

若出现重传，则对传送有用的数据来说，信道的利用率就还要降低。

为了提高传输效率，发送方可以采用流水线传输：发送方可连续发送多个分组，不必每发完一个分组就停下来等待对方的确认。这样可使信道上一直有数据不间断地传送，就可获得很高的信道利用率。




### ⊛ 5.4.2 连续 ARQ 协议

滑动窗口协议比较复杂，是 TCP 协议精髓所在，详细见后续。
发送方维持的发送窗口，意义是：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，这样信道利用率就提高了。

连续 ARQ 协议规定，发送方每收到一个确认，就把发送窗口向前滑动一个分组的位置。如图，发送方收到第 1 个分组的确认，将发送窗口向前移动。


接收方一般采用累积确认的方式：接收方不必对收到的分组逐个发送确认，而是对按序到达的最后一个分组发送确认，这就表示：到这个分组为止的所有分组都已正确收到了。

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。

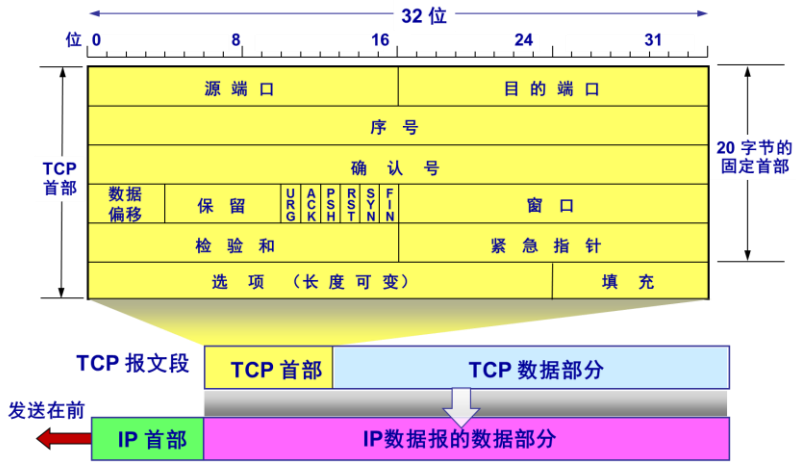
如果发送方发送了前 5 个分组，而中间第 3 个分组丢失了。这时接收方只能对前两个分组发出确认。发送方不知道后三个分组的下落，只好把后面三个分组都再重传一次。这叫做 Go-back-N (回退 N)，表示需要再退回来重传已发送过的 N 个分组。可见当通信线路质量不好时，连续 ARQ 协议会带来负面的影响。

## 🕮 5.5 TCP 报文段的首部格式

TCP 虽然是面向字节流的，但 TCP 传送的数据单元却是报文段。
一个 TCP 报文段分为首部和数据两部分，而 TCP 的全部功能都体现在它首部中各字段的作用。

TCP 报文段首部的前 20 字节是固定的，后面有$4n$字节是根据需要而增加的选项。因此 TCP 首部的最小长度是 20 字节。

如图是 TCP 报文段的首部格式：




(1) 源端口和目的端口：各占 2 字节。TCP 分用功能也是通过端口实现。
(2) 序号：占 4 字节。范围$[0, 2^{32}-1]$，序号使用 mod $2^{32}$运算。
一个 TCP 连接中传送的字节流中的每一个字节都按顺序编号。整个要传送的字节流的起始序号必须在建立连接时设置。首部的序号字段值是本报文段所发送的数据第 1 个字节的序号。
如一个报文段的序号字段是 301，携带数据 100 字节，表明本报文段数据最后一个字节序号为 400；下一个报文段的序号字段值应为 401。

(3) 确认号：占 4 字节，是期望收到对方的下一个报文段数据的第一个字节的序号。如 B 正确收到 A 发来的报文段，序号字段为 501，数据长度 200 字节；因此 B 期望收到 A 的下一个数据序号是 701。所以 B 在发送给 A 的确认报文段中把确认号设置为 701。

若确认号$= N$，表明到序号$N-1$为止的所有数据都已正确收到。

序号字段 32 位，可对 4GB 数据进行编号。

(4) 数据偏移(即首部长度)：占 4 位，指出 TCP 报文段的数据起始处距离 TCP 报文段的起始处有多远。数据偏移的单位是 32 位(4 字节)，4 位二进制最大为 15，因此 TCP 首部最大值为 60 字节。

(5) 保留：占 6 位，保留为今后使用，目前应置为 0。

(6)~(11)是 6 个控制位，各占 1 位
(6) 紧急 URG (URGent)：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(相当于高优先级的数据)。
发送方 TCP 将紧急数据插入到本报文段数据最前面，后面的仍是普通数据。这时需要与首部的紧急指针字段配合使用。

(7) 确认 ACK (ACKnowledgement)：只有当 ACK = 1 时确认号字段才有效。当 ACK = 0 时，确认号无效。TCP 规定在连接建立后所有传送的报文段必须将 ACK 置为 1。

(8) 推送 PSH (PuSH)：有时一端的应用进程希望在键入一个命令后立即收到对方的响应。此时 TCP 就可以使用推送(push)操作。接收方 TCP 收到 PSH = 1 的报文段，就尽快地交付接收应用进程，而不再等到整个缓存都填满了后再向上交付。然而推送操作很少使用。

(9) 复位 RST (ReSeT)：当 RST = 1 时，表明 TCP 连接中出现严重差错(如主机崩溃或其他)，必须释放连接，然后再重新建立运输连接。RST 置为 1 还可以用来拒绝一个非法报文段或拒绝打开一个连接。

(10) 同步 SYN (SYNchronization)：在连接建立时用来同步序号。当 SYN = 1 而 ACK = 0 时，表明是一个连接请求报文段。对方若同意建立连接，应在响应报文段使 SYN =1 和 ACK =1。因此，SYN 置为 1 表示这是一个连接请求或连接接受报文。

(11) 终止 FIN (FINish)：用来释放一个连接。FIN = 1 表明此报文段的发送方的数据已发送完毕，并要求释放运输连接。

---

(12) 窗口：占 2 字节，窗口值[0, 65535]。窗口指发送确认报文段的接收方的接收窗口；窗口值告诉发送方：从本报文段首部中确认号算起，接收方目前允许对方发送的数据量(单位字节)，因为接收方的数据缓存空间有限。总之，窗口值是接收方让发送方设置其发送窗口的依据。
如接收方发送报文段，确认号是 701，窗口字段是 1000，表明从 701 号算起，接收缓存空间还可接收 1000 个字节数据(字节序号 701~1700)。

| 窗口字段明确指出现在允许对方发送的数据量。窗口值经常动态变化。 |
|---|

(13) 检验和：占 2 字节。检验和字段检验的范围包括首部和数据两部分。和 UDP 一样，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。只不过 TCP 的协议字段是 6。

(14) 紧急指针：占 2 字节，仅在 URG =1 时有意义，指出本报文段中紧急数据的字节数(紧急数据结束后是普通数据)。紧急指针指出了紧急数据的末尾在报文段中的位置。即使窗口为 0 时也可以发送紧急数据。

(15) 选项：长度可变，最长 40 字节，没有选项时，TCP 首部 20 字节。
(16) 填充：为了使整个首部长度是 4 字节的整数倍。

**选项**

① TCP 最初只规定了一种选项：最大报文段长度 MSS (Maximum Segment Size)。MSS 是 TCP 报文段中的**数据字段**的最大长度。

MSS 与接收窗口值没有关系。TCP 报文段的数据部分，至少要加上 40 字节的首部(TCP 首部、IP 首部各 20 字节)才能组装成 IP 数据报。
1) 若选择较小的 MSS 长度，网络的利用率就降低。极端情况，如当 TCP 报文段只含有 1 字节的数据时，在 IP 层传输的数据报的开销至少有 40 字节。这样网络

的利用率不会超过 1/41。
2) 若 TCP 报文段非常长，那么在 IP 层传输时就有可能要分成多个短**数据报片**。在终点要把收到的各个短数据报片装配成原来的 TCP 报文段。当传输出错时还要进行重传，这些也都会使开销增大。

因此 MSS 应尽可能大些，只要在 IP 层传输时不需要再分片就行。
由于 IP 数据报所经历的路径是动态变化的，因此在这条路径确定的不需要分片的 MSS，如果改走另一条路径就可能需要分片，故最佳的 MSS 很难确定。传送双方可以有不同的 MSS 值，默认 536 字节。

② 窗口扩大选项：占 3 字节，其中有一个字节表示移位值 S。新的窗口值等于 TCP 首部中的窗口位数增大到(16 + S)。移位值最大 14，相当于窗口最大值左移 14 位，即$2^{16+14}-1=2^{30}-1$。此选项可在双方初始建立 TCP 连接时协商。

③ 时间戳选项：占 10 字节，其中最主要的字段是时间戳值字段(4 字节)和时间戳回送回答字段(4 字节)。该选项两个功能：
1) 计算往返时间 RTT (后续 5.6.2)。
2) 用于处理 TCP 序号超过$2^{32}$的情况，也称防止序号绕回 PAWS (Protect Against Wrapped Sequence numbers)。当使用高速网络时，一次 TCP 连接的数据传送序号很可能被重复使用。如 1.5 Mbit/s 的速率发送报文段时，序号重复要 > 6 h；但用 2.5 Gbit/s 的速率发送，不到 14 s 就会重复。为了使接收方能将新的报文段和迟到的报文段区分，可以在报文段中加上此种时间戳。

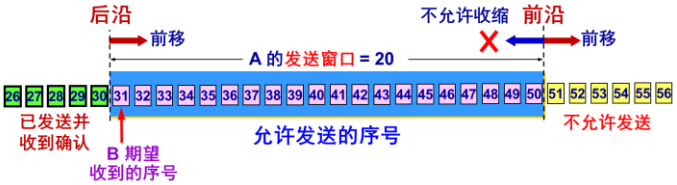
④ 选择确认选项：后续 5.6.3。

**20181113**

## 🕮 5.6 TCP 可靠传输的实现

### ❁ 5.6.1 以字节为单位的滑动窗口

TCP 的滑动窗口是以字节为单位的。
假定 A 收到 B 发来的确认报文段，其中窗口是 20 字节，而确认号是 31，表明 B 期望收到下一个序号是 31，而序号 30 为止的数据已经收到了。
根据 B 给出的窗口值，A 构造出自己的发送窗口，如图：



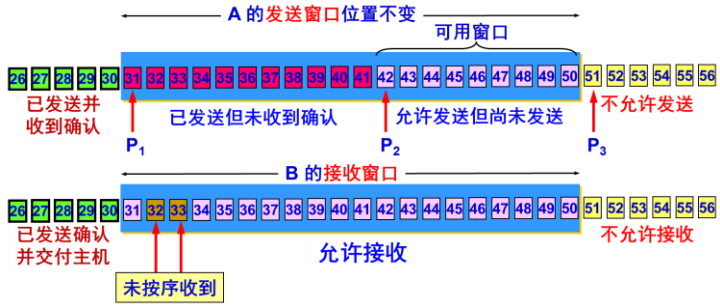

发送窗口表示：在没有收到 B 确认时，A 可以连续把窗口内的数据都发送出去。凡是已经发送的数据，未收到确认前都必须暂时保留，以便超时重传。

显然，窗口越大，发送方就可以在收到对方确认前连续发送更多的数据，因而可能获得更高的传输效率。A 的发送窗口不能超过 B 的接收窗口数值；还受到网络拥塞程度的制约。

收到新确认，发送窗口前沿和后沿都**前移**；没收到新确认，都**不动**。如果新确认通知窗口变小，前沿可能不动或向后收缩，但 TCP 标准**强烈不赞成**收缩。

---

假设 A 发送了序号为 31~41 的数据，如图所示，发送窗口位置未变，表示还未收到确认；靠前的 42~50 字节表示允许发送但还未发送。



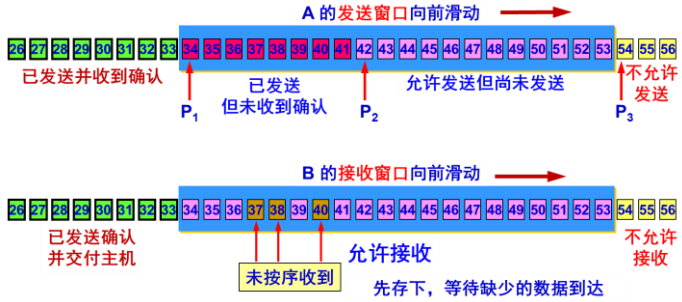

可以看出，描述一个发送窗口的状态需要 3 个指针：

1) $P_3 - P_1$ = A 的发送窗口
2) $P_2 - P_1$ = 已发送但尚未收到确认的字节数
3) $P_3 - P_2$ = 允许发送但尚未发送的字节数(又称可用窗口或有效窗口)

B 的接收窗口大小是 20，接收窗口外到 30 为止的数据是已经发送确认并交付主机的，故 B 没必要再保留了。接收窗口内 31~50 是允许接收的，图中数据没有按序到达，32 和 33 是接收的数据，但 31 还没到(滞留或丢失)。因为 B 只能对最高序号给出确认，因此 B 发送的确认号仍是 31，即期望收到的序号。

---

假设 B 收到序号 31 数据，将 31~33 数据交付主机。B 将接收窗口前移 3 个序号，并向 A 发送确认，窗口值仍是 20，确认号是 34 (虽然收到 37, 38, 40)。




A 收到确认，将发送窗口向前滑动 3 个序号，A 的可用窗口增大。

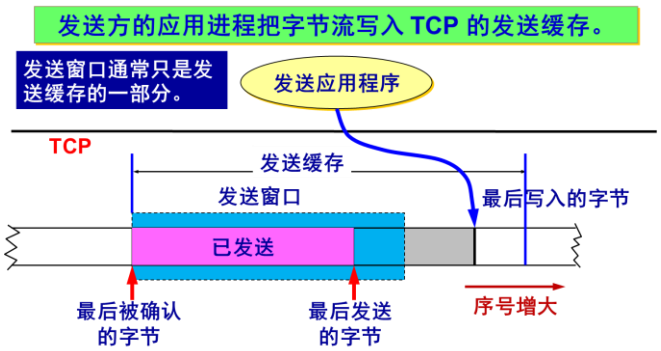
---

A 继续发送 42~53 数据，指针 $P_2$ 前移和 $P_3$ 重合：


此时 A 的发送窗口已满，可用窗口减小为 0，必须停止发送，等待确认。
此时 B 可能已经收到且发送确认，但确认滞留在网络；如果超时，A 需要重传这部分数据，重置超时计时器，直至收到 B 的确认。

**窗口和缓存**

① 发送方维持的发送缓存和发送窗口



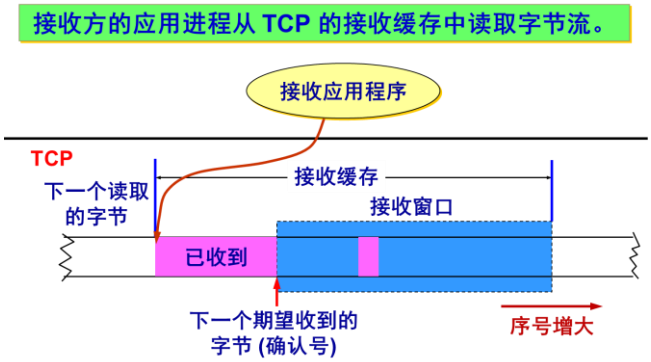

发送缓存暂时存放：

1) 发送方应用程序传给 TCP 准备发送的数据；

2) TCP 已发送但未收到确认的数据。

发送缓存和发送窗口后沿重合。发送应用程序必须控制写入缓存的速率。

② 接收方维持的接收缓存和接收窗口




接收缓存暂时存放：

1) 按序到达、但尚未被接收应用程序读取的数据；

2) 未按序到达的数据。

如果接收应用程序来不及读取收到数据，接收缓存将满，接收窗口减小为 0；反正，接收窗口增大，但不能超过接收缓存大小。

**注意：**

1) A 的发送窗口并不总是和 B 的接收窗口一样大，因为有一定的滞后；还受网络的拥塞情况影响。

2) TCP 标准没有规定对不按序到达的数据如何处理。通常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。

3) TCP 要求接收方必须有累积确认的功能，这样可以减小传输开销。接收方可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。但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。TCP 标准规定，确认推迟不应超过 0.5s。

4) TCP 是全双工通信，每一方都有接收窗口和发送窗口。

### ֍ 5.6.2 超时重传时间的选择

超时重传：TCP 每发送一个报文段，就对这个报文段设置一次计时器。只要计时器设置的重传时间到了但还没有收到确认，就要重传该报文段。
超时重传概念很简单，但是重传时间的选择却是 TCP 最复杂问题之一。

由于 TCP 下层是一个互联网环境，可以只经过一个高速局域网，也可能经过多个低速网络，且每个 IP 数据报所选择的路由可能不同。

如果超时重传时间设置得太短，就会产生很多不必要的重传，使网络负荷增大；若过长，则又使网络的空闲时间增大，降低了传输效率。

TCP 采用一种自适应算法：记录一个报文段发出的时间，以及收到相应确认的时间。两个时间之差就是报文段的往返时间 RTT。TCP 保留了 RTT 的一个加权平均往返时间$RTT_S$ (又称平滑的往返时间 Smoothed Round-Trip Time)。
第一次测量到 RTT 样本时，$RTT_S$值取为所测量到的 RTT 样本值。以后每测量到一个新的 RTT 样本，就重新计算一次$RTT_S$：

$$new\ RTT_S = (1-\alpha) \times (old\ RTT_S) + \alpha \times (new\ RTT\ sample)$$

其中$0 \le \alpha < 1$。若$\alpha \to 0$表示 RTT 值更新较慢。若$\alpha \to 1$表示 RTT 值更新较快。
$\alpha$的推荐值为 $1/8$。

超时重传时间 RTO (Retransmission Time-Out)应略大于上面得出的加权平均往返时间$RTT_S$。
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，与$RTT_S$和新的 RTT 样本之差有关。

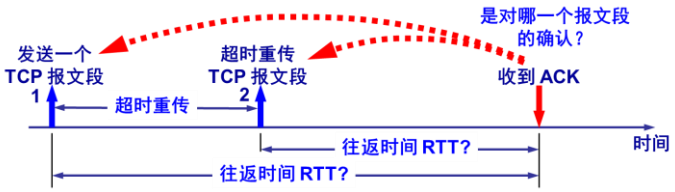
计算 RTTD 建议方法：第一次测量时，$RTT_D$值取为测量到 RTT 样本值的一半。在以后的测量中，则使用下式计算加权平均的$RTT_D$：

$$new\ RTT_D = (1-\beta) \times (old\ RTT_D) + \beta \times |RTT_S - new\ RTT\ sample|$$

$\beta$是小于 1 的系数，推荐值为 $1/4$。

---

如图，发送一个报文段，没有收到确认，重传。过一段时间，收到了确认报文段。那么问题来了：如何判定此确认报文段是对原来的报文段 1 的确认，还是对重传报文段 2 的确认?




Karn 提出一个算法：在计算加权平均$RTT_S$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$RTT_S$和 RTO 就较准确。

但是当报文段的时延突然增大很多，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，超时重传时间就无法更新。

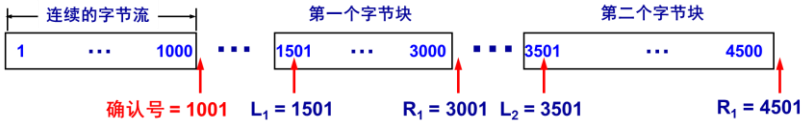
对 Karn 算法修正：报文段每重传一次，就把超时重传时间 RTO 增大一些。典型的做法是取新重传时间的 2 倍。当不再发生报文段的重传时，才根据之前公式计算超时重传时间。

### ֍ 5.6.3 选择确认 SACK

若收到的报文段无差错，只是未按序号，中间还缺少一些序号的数据，那么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?

选择确认(Selective ACK)就是一种可行的处理方法。

如图，TCP 接收方在收到的数据字节流的序号不连续，形成了一些不连续的字节块。如果这些字节的序号都在接收窗口内，那么先收下这些数据，但要把这些信息准确告诉发送方，使其不要再重复发送已收到的数据。




和前后字节不连续的每一个字节块都有两个边界：左边界和右边界。

字节块格式为左闭右开：第 1 个字节块区间为$[L_1, R_1) = [1501, 3001)$；第 2 个字节块区间为$[L_2, R_2) = [3501, 4501)$。

RFC 2018 规定：

> 如果要使用选择确认，那么在建立 TCP 连接时，就要在 TCP 首部的选项中加上"允许 SACK"的选项，而双方必须都事先商定好。
>
> 如果使用选择确认，那么原来首部中的"确认号字段"用法仍然不变。只是以后在 TCP 报文段的首部中都增加了 SACK 选项，以便报告收到的不连续的字节块的边界。由于首部选项长度最多只有 40 字节，而指明 1 个边界就要 4 字节 (因为序号 32 位)，4 个字节块(8 个边界)需要 32 字节；另外需要 1 个字节指明使用 SACK 选项；1 个字节指明 SACK 选项占多少字节。因此总共需 34 字节。若报告 5 个字节块的边界信息需要 42 字节，超过 40 字节上限。

然而 SACK 文档并没有指明发送方应怎样响应 SACK，大多数的实现还是**重传所有**未确认的数据块。

## 🕮 5.7 TCP 流量控制

### ֍ 5.7.1 利用滑动窗口实现流量控制

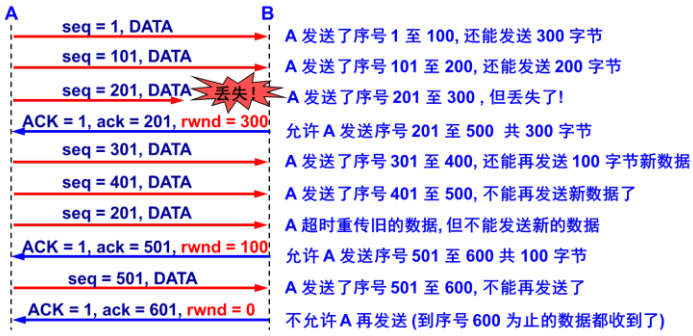
流量控制 (flow control)是让发送方发送速率不要太快，让接收方来得及接收，防止数据丢失。

利用滑动窗口机制可以很方便地在 TCP 连接上实现流量控制。

如图是利用滑动窗口机制进行流量控制的举例：

设 A 向 B 发送数据。在连接建立时，B 告诉 A："我的接收窗口 rwnd = 400"。rwnd 表示接收窗口(receive window)，单位为字节。因此 A 的发送窗口不能超过

B 给出的接收窗口的数值。
假设每个报文段 100 字节，初始序号为 1 (图中的 seq = 1)。
**注意**：图中 ACK 表示首部中确认位；ack 表示确认字段的值。




接收方 B 进行了三次流量控制：rwnd = 300 → 100 → 0。为 0 表示 A 不能再发送数据了。直到 B 重新发送一个新的窗口值前，A 将处于暂停发送状态。

---

假设此后 B 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。若此报文段在传送过程丢失了，A 将一直等待收到 B 发送非 0 窗口的通知，而 B 也一直等待 A 发送数据，造成死锁(deadlock)。

为了解决这个问题，TCP 为每个连接设有一个持续计时器(persistence timer)。
只要 TCP 连接的一方(发送方)收到对方(接收方)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。
1) 若窗口仍是零，则收到这个报文段的一方(发送方)就重新设置持续计时器。
2) 若窗口非 0，则死锁的僵局就可以打破。

**注**：TCP 规定，即使设为零窗口，也必须接收：**零窗口探测报文段、确认报文段、携带紧急数据的报文段**。

### 5.7.2 TCP 的传输效率

可以用不同的机制控制 TCP 报文段的发送时机：
1) TCP 维持一个变量，它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) 由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) 发送方的一个计时器期限到了，这时就把当前已有的缓存数据装入报文段(但不能超过 MSS)发送出去。

如何控制 TCP 发送报文段的时机仍然是一个较为复杂的问题。
若发送方只发送 1 字节的数据，需要构成 41 字节的 IP 数据报；接收方发出确认，构成的 IP 数据报 40 字节(假设没有数据)。如此传送效率太低，因此应适当推迟发回确认报文，尽量使用捎带确认的方法。

---

TCP 广泛使用 Nagle 算法：
1) 若发送应用进程把要发送的数据逐个字节地送到 TCP 的发送缓存，则发送

方就把第一个字节先发送出去，把后面到达的数据字节都缓存起来。
2) 当发送方收到对第一个字节的确认后，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。
3) 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。
4) 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，能有效提高网络的吞吐量。

---

糊涂窗口综合症(silly window syndrome)
TCP 接收方缓存已满，会向发送方发送零窗口报文。若接收方的应用进程以交互方式每次只读取 1 个字节，于是缓存空间腾出 1 字节，接收方又发送窗口大小为 1 字节的更新报文(IP 数据报 40 字节)，发送方应邀发送 1 字节的数据(IP 数据报 41 字节)；然后接收方缓存又满了，如此循环往复，使得网络效率很低。

解决方法：让接收方等待一段时间，使得接收缓存已有足够空间容纳一个最长的报文段，或等到接收缓存已有一半空闲的空间。只要出现这两种情况之一，接收方就发出确认报文，并向发送方通知当前的窗口大小。

---

因此：发送方不要发送很小的报文段；接收方也不要在缓存刚有一小点空间时就将这个很小的窗口大小信息通知给发送方。

**20181116**

## 🕮 5.8 TCP 的拥塞控制

### ❁ 5.8.1 拥塞控制的一般原理

计算机网络中的链路容量(带宽)、交换结点中的缓存和处理机等，都是网络资源。在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。这种现象称为拥塞(congestion)。
出现拥塞的原因：

∑对资源需求 > 可用资源

若网络中有许多资源同时产生拥塞，网络性能就会明显变坏，整个网络的吞吐量将随输入负荷的增大而下降。

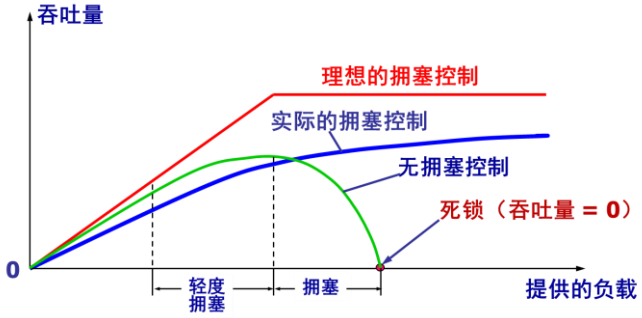
增加资源能解决拥塞吗？
不能。因为网络拥塞是一个非常复杂的问题，是由许多因素引起的。简单地增加资源，很多时候不但不能解决拥塞问题，而且还可能使网络的性能更坏。

**拥塞控制与流量控制的区别**

1) 拥塞控制是防止过多的数据注入到网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制前提：网络能够承受现有的网络负荷。
拥塞控制是一个全局性的过程，涉及到所有的主机、所有的路由器，以及与降低网络传输性能有关的所有因素。TCP 端点只有迟迟收不到确认，就认为网络中某处可能发生了拥塞，但无法知道拥塞在何处，也无法知道原因。
2) 流量控制是一个端到端的问题，是接收端抑制发送端发送数据的速率，以便接收端来得及接收。

拥塞控制和流量控制之所以常常被弄混，是因为某些拥塞控制算法是向发送端发送控制报文，并告诉发送端，网络已出现麻烦，必须放慢发送速率。这点又和流量控制是很相似的。

如图，横坐标是提供的负载(offered load)，代表单位时间内输入网络的分组数目，也称输入负载或网络负载。纵坐标是吞吐量(throughput)，代表单位时间内从网络输出的分组数目。




实际网络，随着提供的负载增大，网络吞吐量增长速率减小；当提供的负载达到某一数值，网络的吞吐量反而下降，进入拥塞状态；当继续增大，网络的吞吐量下降到 0，网络无法工作，进入死锁(deadlock)状态。

实践证明，拥塞控制是很难设计的，因为它是一个动态问题。当前网络正朝着高速化发展，这很容易出现缓存不够大而造成分组丢失。但分组的丢失是网络发生拥塞的征兆而不是原因。许多情况下，甚至正是**拥塞控制**本身成为引起网络性能恶化甚至发生死锁的原因。

拥塞控制方法分为：

① 开环控制

设计网络时事先将有关发生拥塞的因素考虑周到，力求网络在工作时不产生拥塞。一旦整个系统运行起来，就不再改正。

② 闭环控制

基于反馈环路的概念。主要的几种措施：

1) 监测网络系统以便检测到拥塞在何时、何处发生。
2) 将拥塞发生的信息传送到可采取行动的地方。
3) 调整网络系统的运行以解决出现的问题。

拥塞的指标主要有：

1) 由于缺少缓存空间而被丢弃的分组的百分数；
2) 平均队列长度；
3) 超时重传的分组数；
4) 平均分组时延；
5) 分组时延的标准差...

一般监测到拥塞发生时，要将拥塞发生的信息传送到源站，但是通知拥塞的分组也会使网络更加拥塞。另一种方法是在路由器转发的分组中保留一个比特或字段，用于表示有没有产生拥塞。也可由一些主机或路由器周期性发出探测分组，询问

拥塞是否发生。

过于频繁采取行动来缓解网络拥塞，会使系统产生不稳定的振荡；但过于迟缓采取行动又没有任何实用价值。因此选择正确的时间是相当困难的。

### ֍ 5.8.2 TCP 的拥塞控制方法

TCP 采用**基于窗口**的拥塞控制，该方法属于闭环控制方法。
为了进行拥塞控制，TCP 发送方要维持一个拥塞窗口 cwnd (congestion window)的状态变量，大小取决于网络的拥塞程度，并动态变化。
发送方让自己的发送窗口取为**拥塞窗口**和接收方的**接收窗口**较小值。

发送方控制拥塞窗口的原则：
1) 只要网络没有出现拥塞，拥塞窗口就可以再增大一些，以便把更多的分组发送出去，这样就可以提高网络的利用率。
2) 但只要网络出现拥塞或有可能出现拥塞，就必须把拥塞窗口减小一些，以减少注入到网络中的分组数，以便缓解网络出现的拥塞。

拥塞的判断：没有按时收到确认报文
现在通信线路的传输质量一般都很好，因传输出差错而丢弃分组的概率很小(远小于1%)。因此只要出现了**超时**，就可以判断网络出现了拥塞。

TCP 的拥塞控制采用 4 种算法：慢开始(slow-start)、拥塞避免(congestion avoidance)、快重传(fast retransmit)、快恢复(fast recovery)。

#### ① 慢开始

用来确定网络的负载能力。
算法的思路：由小到大逐渐增大拥塞窗口数值。
初始拥塞窗口 cwnd 设置：
旧的规定：在刚刚开始发送报文段时，先把初始拥塞窗口 cwnd 设置为 1 至 2 个发送方的最大报文段 SMSS (Sender Maximum Segment Size) 的数值。
新的 RFC 5681 把初始拥塞窗口 cwnd 设置为不超过 2 至 4 个 SMSS 的数值。
慢开始门限 ssthresh（状态变量）：防止拥塞窗口 cwnd 增长过大引起网络拥塞。
拥塞窗口 cwnd 每次的增加量 = min (N, SMSS)　　(5-8)

发送端利用拥塞窗口根据网络的拥塞情况调整发送的数据量。
所以，发送窗口大小不仅取决于接收方公告的接收窗口，还取决于网络的拥塞状况，所以真正的发送窗口值为：
真正的发送窗口值 = Min(公告窗口值，拥塞窗口值)

### ֍ 5.8.3 主动队列管理 AQM

在网络层，也可以使路由器采用适当的分组丢弃策略(如主动队列管理 AQM)减少网络拥塞的发生。

## 🕮 5.9 TCP 的运输连接管理

运输连接三个阶段：连接建立、数据传送、连接释放

### ֍ 5.9.1 TCP 的连接建立

主动发起 TCP 连接建立的应用进程称为客户；被动等待连接建立的应用进程称为服务器。TCP 连接建立采用三报文握手机制。服务器要确认客户的连接请求，然后客户要对服务器的确认进行确认。

### ֍ 5.9.2 TCP 的连接释放

TCP 的连接释放采用四报文握手机制。任何一方都可以在数据传送结束后发出连接释放的通知，待对方确认后进入半关闭状态。当另一方也没数据再发送时，则发送连接释放通知，对方确认就完全关闭 TCP 连接。

### ֍ 5.9.3 TCP 的有限状态机