计算机网络 - 传输层

UDP

网络层只把分组发送到目的主机，但是真正通信的并不是主机而是主机中的进程。传输层提供了进程间的逻辑通信，传输层向高层用户屏蔽了下面网络层的核心细节，使应用程序看起来像是在两个传输层实体之间有一条端到端的逻辑通信信道。

UDP 和 TCP 的特点

- 用户数据报协议 UDP（User Datagram Protocol）是无连接的，尽最大可能交付，没有拥塞控制，面向报文（对于应用程序传下来的报文不合并也不拆分，只是添加 UDP 首部），支持一对一、一对多、多对一和多对多的交互通信。

- 传输控制协议 TCP（Transmission Control Protocol）是面向连接的，提供可靠交付，有流量控制，拥塞控制，提供全双工通信，面向字节流（把应用层传下来的报文看成字节流，把字节流组织成大小不等的数据块），每一条 TCP 连接只能是点对点的（一对一）。
- UDP套接字由一个（目的IP地址，目的端口号）的二元组标识，所有具有相同IP地址和端口号的数据将被送入同一个套接字（即便他们来自不同的服务器），而TCP的套接字由（源IP地址，源端口号，目的IP地址，目的端口号）标识，因此UDP可以多对多，而TCP只能一对一。

UDP 首部格式

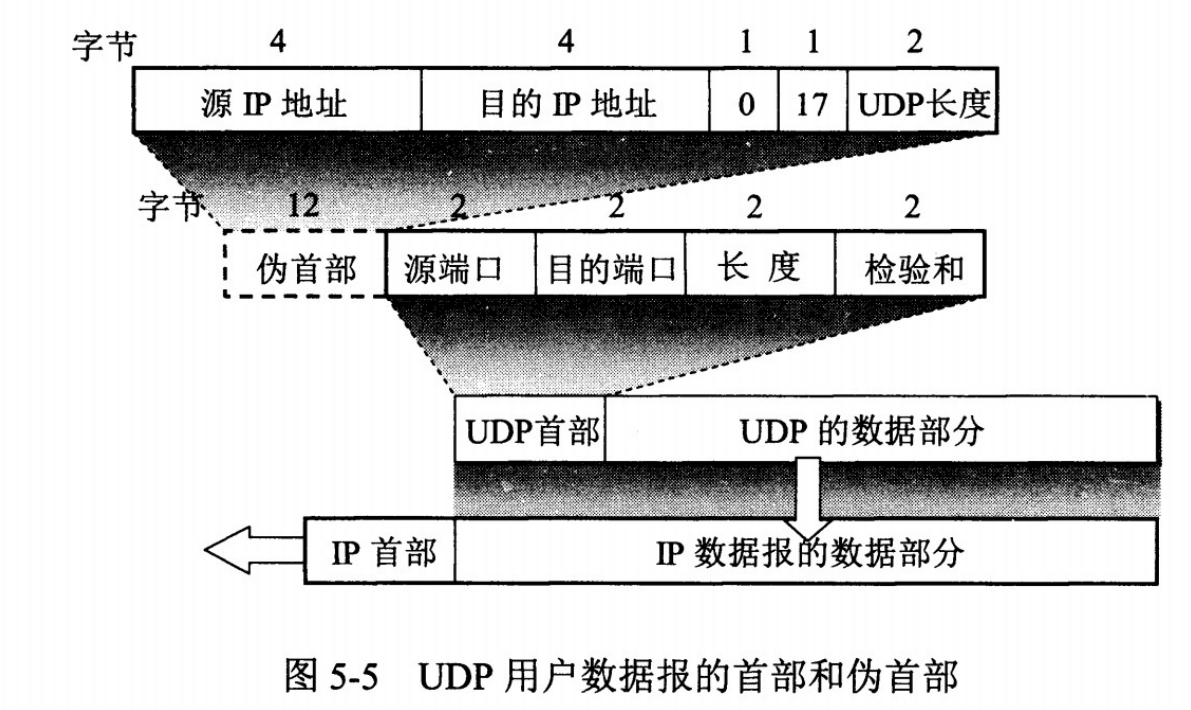


图 5-5　UDP 用户数据报的首部和伪首部

首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

UDP检验和

UDP的检验和用于检验比特是否发生反转，检验和对于所有16 bit（如果数据中不足16 bit会在末尾补0）的字进行求和（最高位进位会被加在最低位上，被称为回卷），然后取反得到检验和。

0110011001100000
0101010101010101
1000111100001100

这些 16 比特字的前两个之和是：

0110011001100000
0101010101010101
1011101110110101

再将上面的和与第三个字相加，得出：

1011101110110101
1000111100001100
0100101011000010

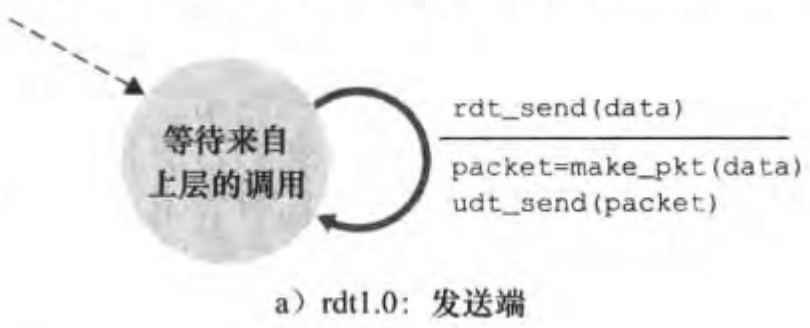
可靠数据传输协议（reliable data transfer）

运输层协议对于底层的数据传输具有以下假定：

- 底层的传输协议与信道是不可靠的点对待你信道，不能保证数据准确传输（IP协议是best effort的）
- 底层信道不会对分组进行重新排序

rdt1.0

假设：底层信道完全可靠，不会发生数据丢失



a）rdt1.0：发送端

b）rdt1.0：接收端

图 3-9　rdt1.0：用于完全可靠信道的协议

那么只用构造上图所示的有限自动机就可以完成数据的分发。

rdt2.0 与rdt2.1 ,rdt2.2

假定：底层信道只发生比特传输错误，不考虑丢包事件。

1. 引入确认机制。接收方在接受文件之后，返回一个肯定或者否定确认。
2. 自动重传协议。在比特出现错误时，支持自动重新传输文件。需要按照1.差错检测 2.接收方反馈3.重传
3. 如何确认当前数据包是上一个的重传还是新文件。rdt2.1引入序号机制，判断当前文件是否为重传。
4. rdt2.2删除了否定确认，利用有限自动机的特性。

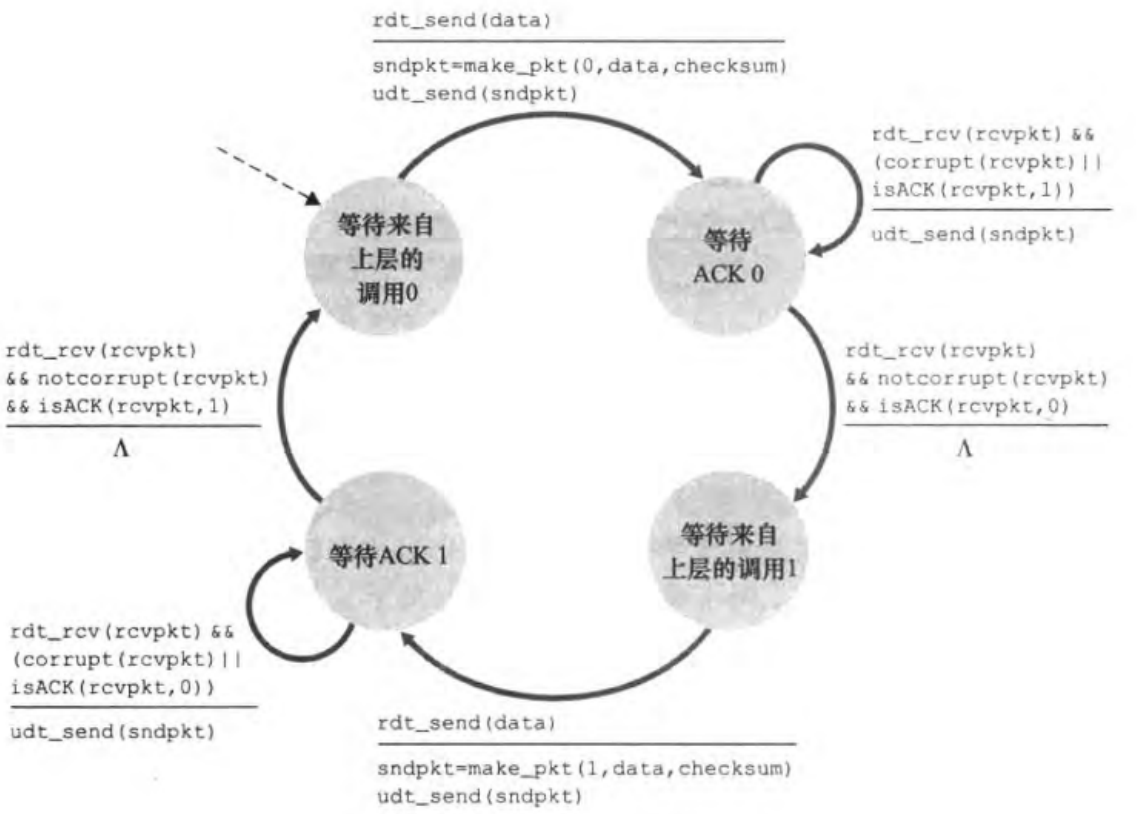


图 3-13　rdt2. 2 发送方

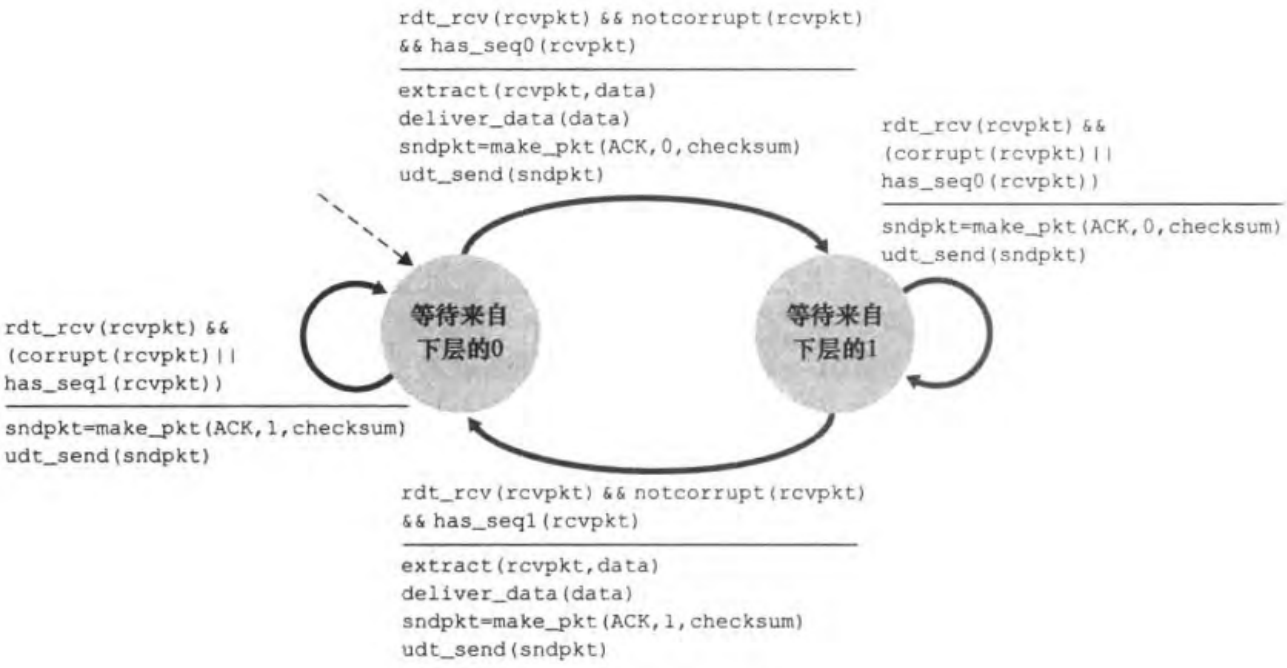


图 3-14　rdt2. 2 接收方

rdt3.0

考虑比特翻转和分组丢失的情况，这和实际情况一致

- 引入了定时器机制来解决分组丢失的问题。当分组发出时就开始定时器，在设定的时间之后没有收到确认报文就重传，并重新设置定时器。

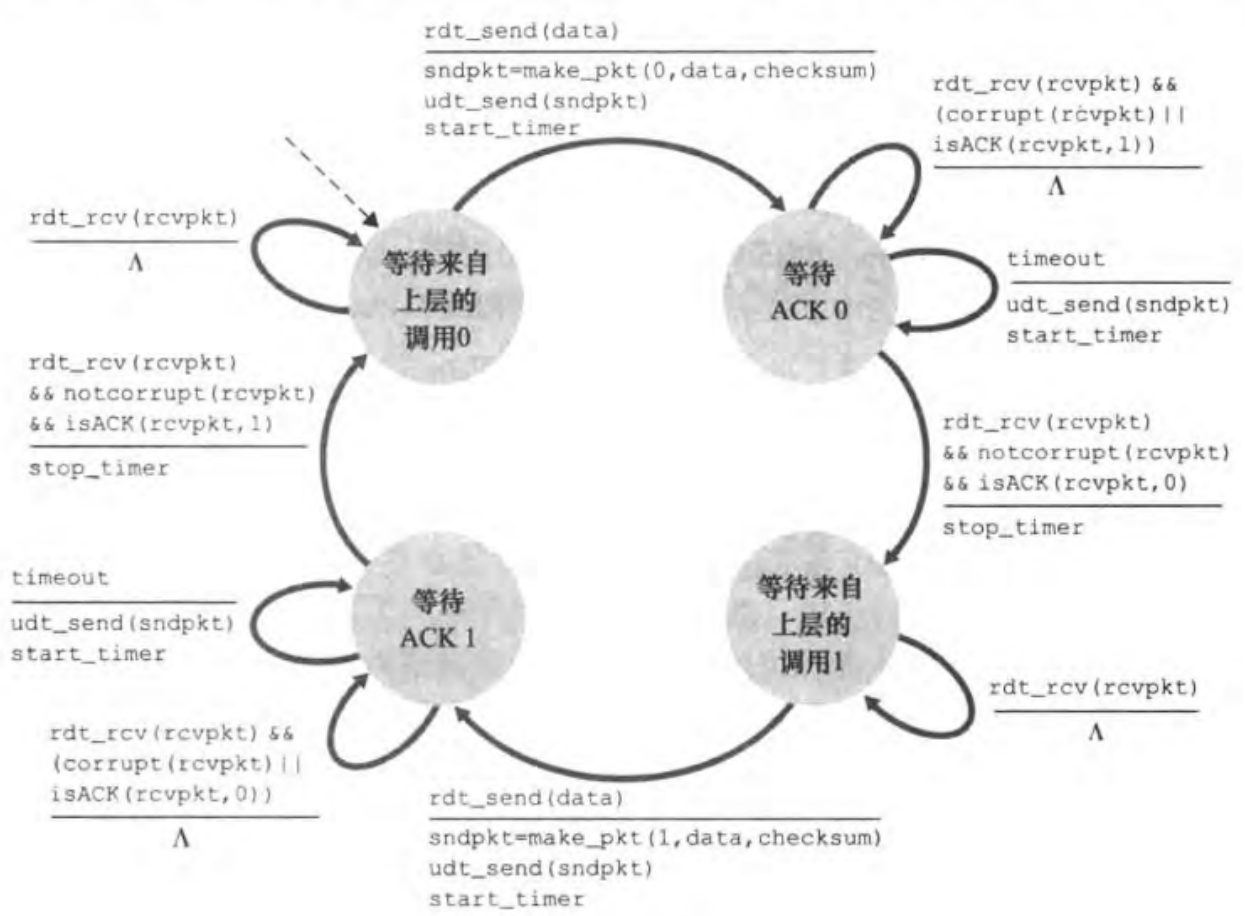


图 3-15　rdt3.0 发送方

流水线传输机制

前面的可靠传输方法都是停等协议，每次需要收到接收方的确认才能发送下一个分组，这也效率很低，所以需要考虑进行流水线处理，提高效率。

滑动窗口和GBN（go-back-N)方法

上面所考虑的都是居于停等的传输协议，传输效率很低。因此，我们采用流水线的方式提高传输效率。 1. 滑动窗口。见TCP滑动窗口。 2. GBN解决滑动窗口的分组丢失问题。

TCP

TCP 首部格式

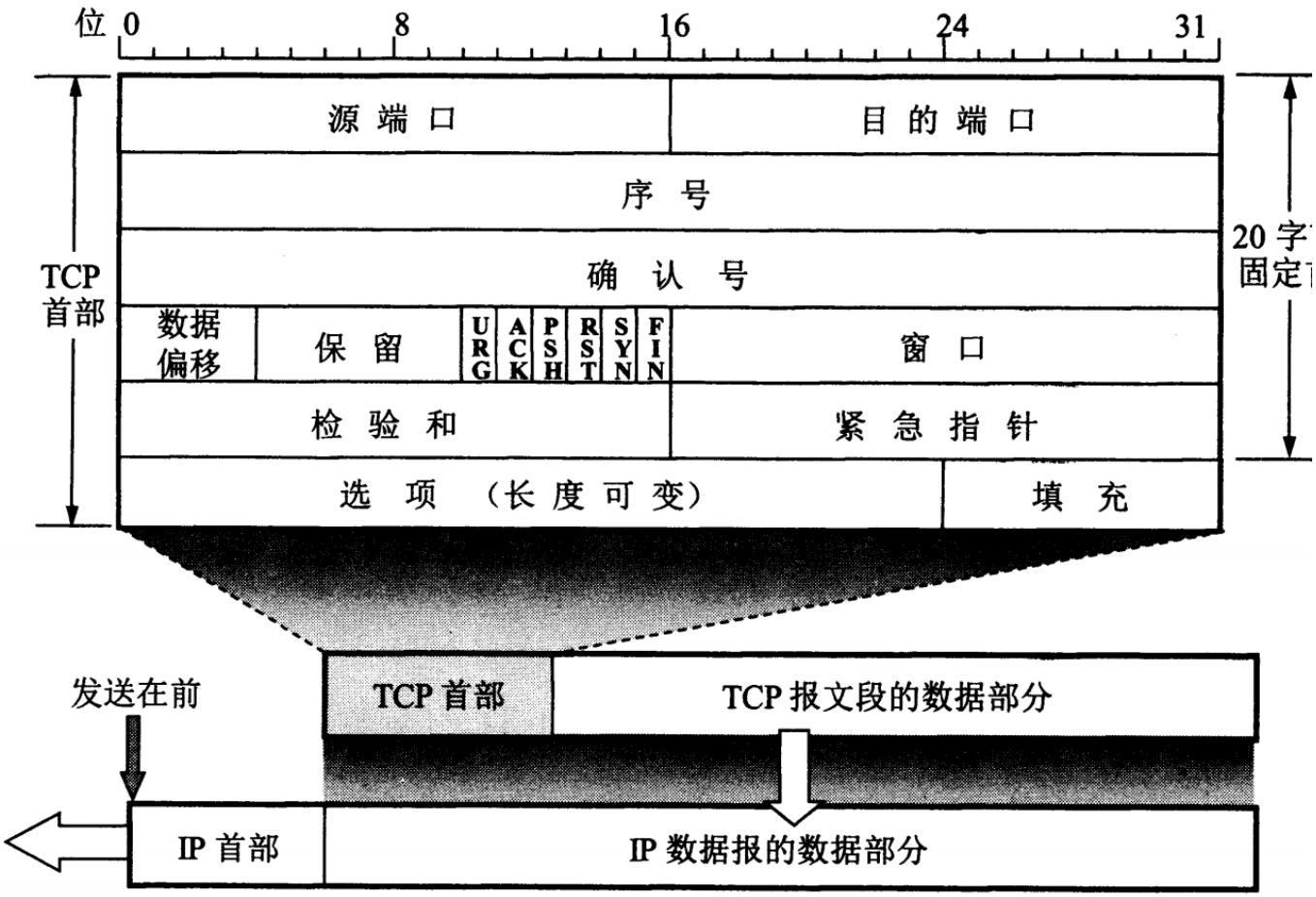


图 5-14　TCP 报文段的首部格式

- **序号**：用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。
- **确认号**：期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。
- **数据偏移**：指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
- **确认 ACK**：当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把 ACK 置 1。
- **同步 SYN**：在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
- **终止 FIN**：用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
- **窗口**：窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。

TCP 的三次握手

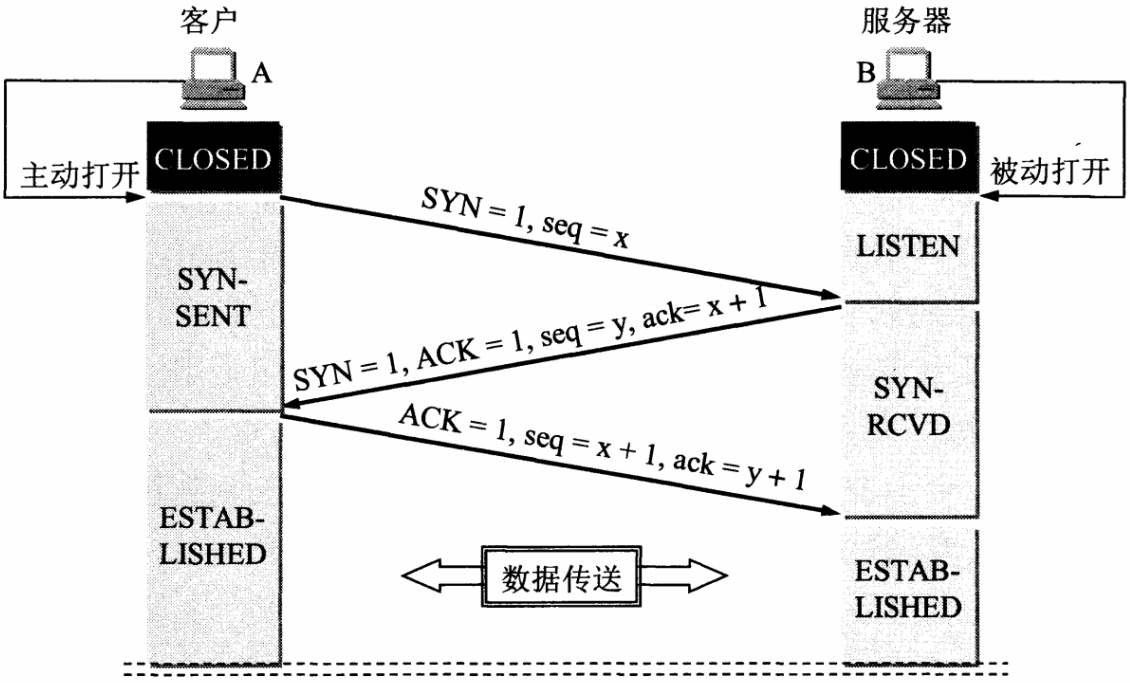


图 5-28 用三报文握手建立 TCP 连接

假设 A 为客户端，B 为服务器端。

- 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
- A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
- B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。
- A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
- B 收到 A 的确认后，连接建立。

三次握手的原因

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

TCP 的四次挥手

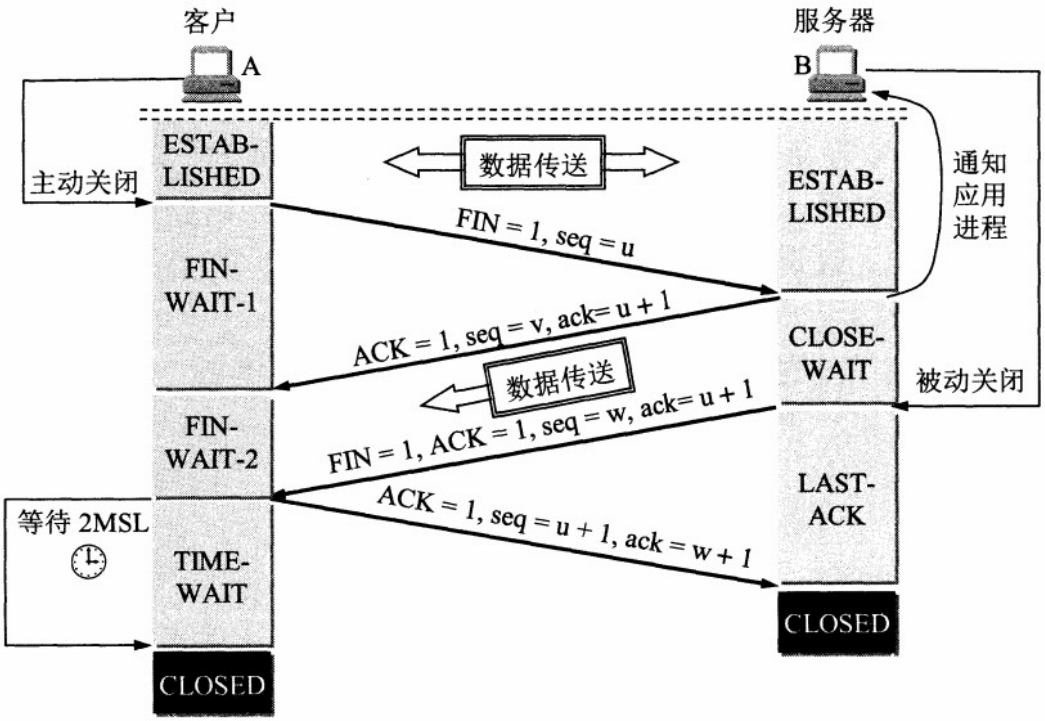


图 5-29　TCP 连接释放的过程

以下描述不讨论序号和确认号，因为序号和确认号的规则比较简单。并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

- A 发送连接释放报文，FIN=1。
- B 收到之后发出确认，此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
- 当 B 不再需要连接时，发送连接释放报文，FIN=1。
- A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。
- B 收到 A 的确认后释放连接。

四次挥手的原因

客户端发送了 FIN 连接释放报文之后，服务器收到了这个报文，就进入了 CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME_WAIT

客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL。这么做有两个理由：

- 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。

- 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

TCP 可靠传输

TCP 使用超时重传来实现可靠传输：如果一个已经发送的报文段在超时时间内没有收到确认，那么就重传这个报文段。

一个报文段从发送再到接收到确认所经过的时间称为往返时间 RTT，加权平均往返时间 RTTs 计算如下：

$$RTTs = (1 - \alpha) * RTTs + \alpha * RTT$$

其中，0 ≤ α < 1，RTTs 随着 α 的增加更容易受到 RTT 的影响。

超时时间 RTO 应该略大于 RTTs，TCP 使用的超时时间计算如下：

$$RTO = RTTs + 4 * RTT_d$$

其中RTT_d为偏差的加权平均值。

$$RTT_d = (1 - \beta) * RTTs + \beta * |(RTT_{real} - RTTs)|$$

TCP 滑动窗口

窗口是缓存的一部分，用来暂时存放字节流。发送方和接收方各有一个窗口，接收方通过 TCP 报文段中的窗口字段告诉发送方自己的窗口大小，发送方根据这个值和其它信息设置自己的窗口大小。

发送窗口内的字节都允许被发送，接收窗口内的字节都允许被接收。如果发送窗口左部的字节已经发送并且收到了确认，那么就将发送窗口向右滑动一定距离，直到左部第一个字节不是已发送并且已确认的状态；接收窗口的滑动类似，接收窗口左部字节已经发送确认并交付主机，就向右滑动接收窗口。

接收窗口只会对窗口内最后一个按序到达的字节进行确认，例如接收窗口已经收到的字节为 {31, 34, 35}，其中 {31} 按序到达，而 {34, 35} 就不是，因此只对字节 31 进行确认。发送方得到一个字节的确认之后，就知道这个字节之前的所有字节都已经被接收。

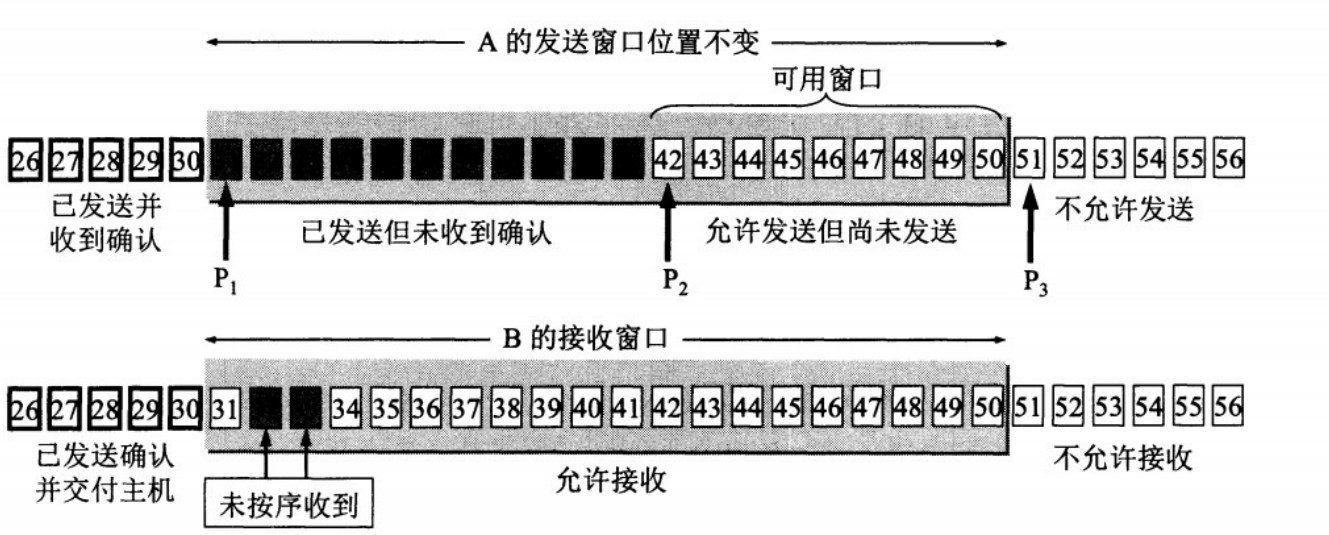


图 5-16　A 发送了 11 个字节的数据

TCP 流量控制

流量控制是为了控制发送方发送速率，保证接收方来得及接收。

接收方发送的确认报文中的窗口字段可以用来控制发送方窗口大小，从而影响发送方的发送速率。将窗口字段设置为 0，则发送方不能发送数据。

TCP 拥塞控制

如果网络出现拥塞，分组将会丢失，此时发送方会继续重传，从而导致网络拥塞程度更高。因此当出现拥塞时，应当控制发送方的速率。这一点和流量控制很像，但是出发点不同。流量控制是为了让接收方能来得及接收，而拥塞控制是为了降低整个网络的拥塞程度。

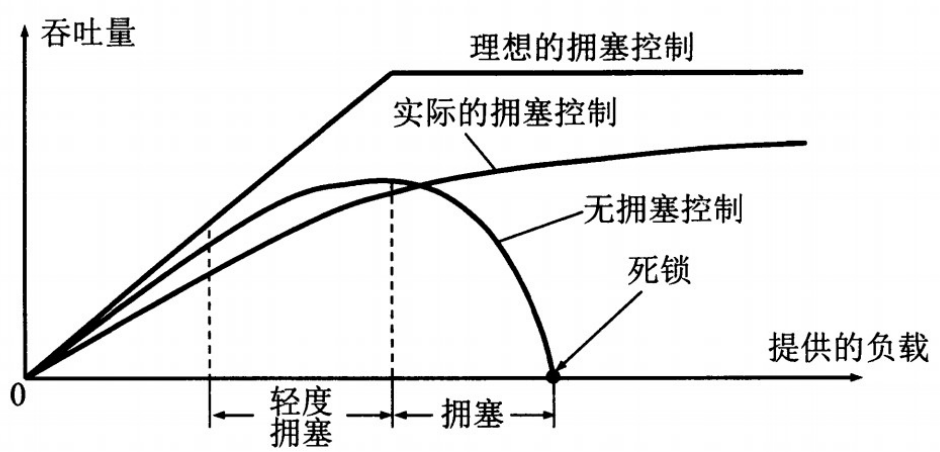


图 5-23 拥塞控制所起的作用

TCP 主要通过四个算法来进行拥塞控制：慢开始、拥塞避免、快重传、快恢复。

发送方需要维护一个叫做拥塞窗口（cwnd）的状态变量，注意拥塞窗口与发送方窗口的区别：拥塞窗口只是一个状态变量，实际决定发送方能发送多少数据的是发送方窗口。

为了便于讨论，做如下假设：

- 接收方有足够大的接收缓存，因此不会发生流量控制；
- 虽然 TCP 的窗口基于字节，但是这里设窗口的大小单位为报文段。

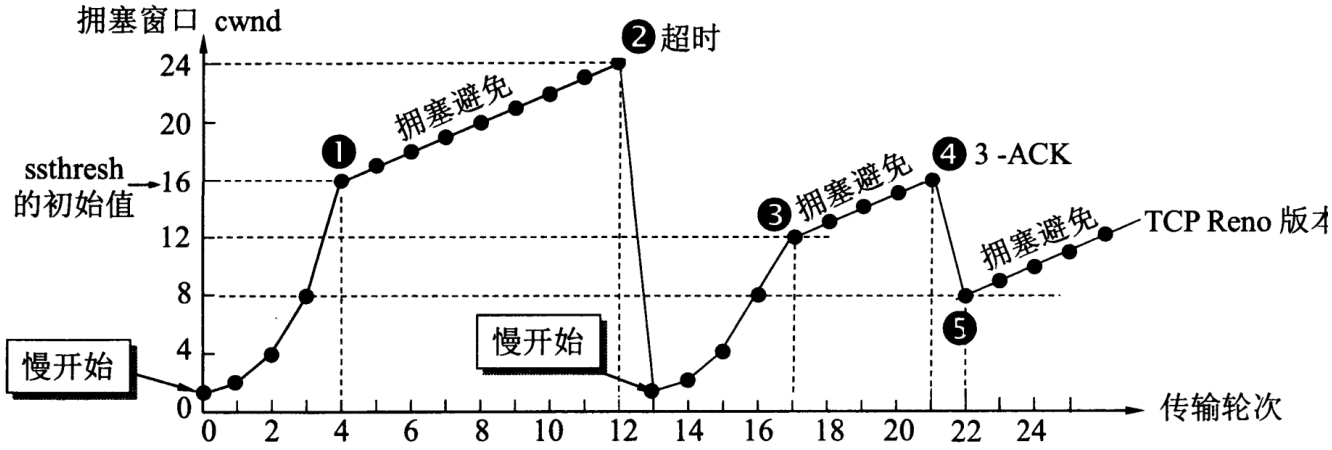


图 5-25　TCP 拥塞窗口 cwnd 在拥塞控制时的变化情况

1. 慢开始与拥塞避免

发送的最初执行慢开始，令 cwnd = 1，发送方只能发送 1 个报文段；当收到确认后，将 cwnd 加倍，因此之后发送方能够发送的报文段数量为：2、4、8 ...

注意到慢开始每个轮次都将 cwnd 加倍，这样会让 cwnd 增长速度非常快，从而使得发送方发送的速度增长速度过快，网络拥塞的可能性也就更高。设置一个慢开始门限 ssthresh，当 cwnd >= ssthresh 时，进入拥塞避免，每个轮次只将 cwnd 加 1。

如果出现了超时，则令 ssthresh = cwnd / 2，然后重新执行慢开始。

2. 快重传与快恢复

在接收方，要求每次接收到报文段都应该对最后一个已收到的有序报文段进行确认。例如已经接收到 M_1 和 M_2 ，此时收到 M_4，应当发送对 M_2 的确认。

在发送方，如果收到三个重复确认，那么可以知道下一个报文段丢失，此时执行快重传，立即重传下一个报文段。例如收到三个 M_2，则 M_3 丢失，立即重传 M_3。

在这种情况下，只是丢失个别报文段，而不是网络拥塞。因此执行快恢复，令 ssthresh = cwnd / 2 ， cwnd = ssthresh，注意到此时直接进入拥塞避免。

慢开始和快恢复的快慢指的是 cwnd 的设定值，而不是 cwnd 的增长速率。慢开始 cwnd 设定为 1，而快恢复 cwnd 设定为 ssthresh。

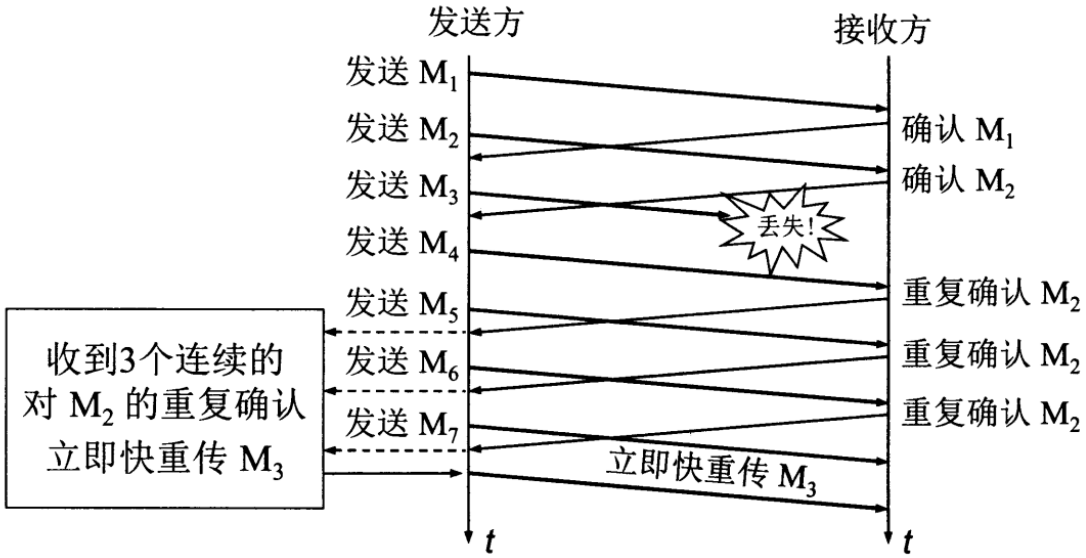


图 5-26　快重传的示意图