# 计网题目

jask

09/09/2024

## 重传机制

TCP 实现可靠传输的方式之一，是通过序列号与确认应答。

在 TCP 中，当发送端的数据到达接收主机时，接收端主机会返回一个确认应答消息，表示已收到消息。

常见的重传机制：

超时重传

快速重传

SACK

D-SACK

### 超时重传

重传机制的其中一个方式，就是在发送数据时，设定一个定时器，当超过指定的时间后，没有收到对方的 ACK 确认应答报文，就会重发该数据，也就是我们常说的超时重传。

TCP 会在以下两种情况发生超时重传：

数据包丢失

确认应答丢失

#### 超时时间应该设置为多少呢？

我们先来了解一下什么是 RTT（Round-Trip Time 往返时延），从下图我们就可以知道：

RTT 就是数据从网络一端传送到另一端所需的时间，也就是包的往返时间。

超时重传时间是以 RTO（Retransmission Timeout 超时重传时间）表示。

#### 假设在重传的情况下，超时时间 RTO 「较长或较短」时，会发生什么事情呢？

当超时时间 RTO 较大时，重发就慢，丢了老半天才重发，没有效率，性能差；

当超时时间 RTO 较小时，会导致可能并没有丢就重发，于是重发的就快，会增加网络拥塞，导致更多的超时，更多的超时导致更多的重发。

根据上述的两种情况，我们可以得知，超时重传时间 RTO 的值应该略大于报文往返 RTT 的值。

实际上「报文往返 RTT 的值」是经常变化的，因为我们的网络也是时常变化的。也就因为「报文往返 RTT 的值」是经常波动变化的，所以「超时重传时间 RTO 的值」应该是一个动态变化的值。

估计往返时间，通常需要采样以下两个：

需要 TCP 通过采样 RTT 的时间，然后进行加权平均，算出一个平滑 RTT 的值，而且这个值还是要不断变化的，因为网络状况不断地变化。

除了采样 RTT，还要采样 RTT 的波动范围，这样就避免如果 RTT 有一个大的波动的话，很难被发现的情况。

其中 SRTT 是计算平滑的 RTT ，DevRTR 是计算平滑的 RTT 与最新 RTT 的差距。

在 Linux 下，$\alpha = 0.125$，$\beta = 0.25$，$\mu = 1$，$\delta = 4$。别问怎么来的，问就是大量实验中调出来的。

如果超时重发的数据，再次超时的时候，又需要重传的时候，TCP 的策略是超时间隔加倍。

也就是每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。

超时触发重传存在的问题是，超时周期可能相对较长。那是不是可以有更快的方式呢？

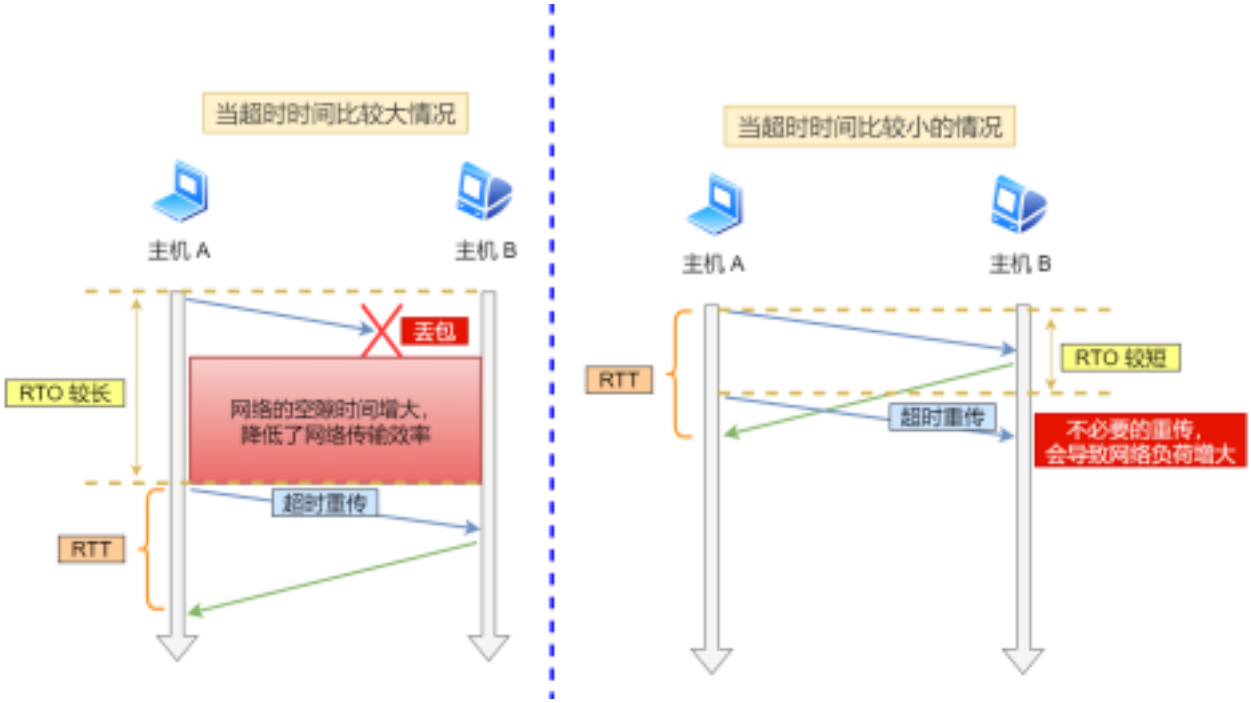



Figure 1: 两种情况

① 首次计算 RTO，其中 R1 为第一次测量的 RTT

$$SRTT = R1$$

$$DevRTT = R1/2$$

$$RTO = \mu * SRTT + \partial * DevRT = \mu * R1 + \partial * (R1/2)$$

② 后续计算 RTO，其中 R2 为最新测量的 RTT

$$SRTT = SRTT + \alpha ( RTT - SRTT ) = R1 + \alpha * ( R2 - R1 )$$

$$DevRTT = ( 1-\beta )*DevRTT + \beta *( |RTT-SRTT| ) =(1-\beta) * ( R1/2 ) + \beta * ( |R2 - R1| )$$

$$RTO = \mu * SRTT + \partial * DevRTT$$

Figure 2: 计算公式

于是就可以用「快速重传」机制来解决超时重发的时间等待

### 快速重传

TCP 还有另外一种快速重传 (Fast Retransmit) 机制，它不以时间为驱动，而是以数据驱动重传。

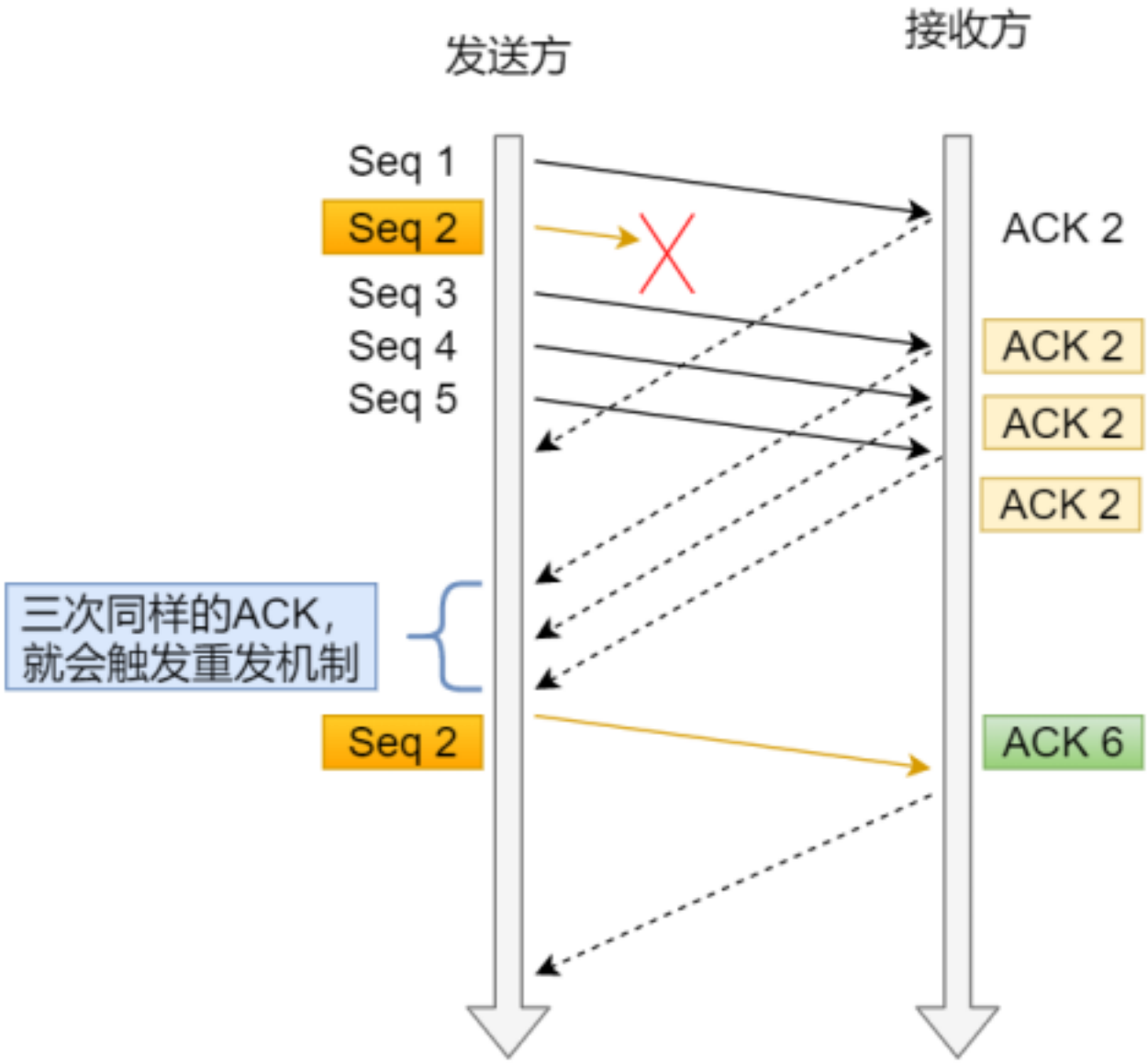



Figure 3: 工作流程

在上图，发送方发出了 1,2,3,4,5 份数据：

第一份 Seq1 先送到了，于是就 Ack 回 2；

结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2；

后面的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到；

发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。

最后，收到了 Seq2，此时因为 Seq3,Seq4,Seq5 都收到了，于是 Ack 回 6 。

所以，快速重传的工作方式是当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。

快速重传机制只解决了一个问题，就是超时时间的问题，但是它依然面临着另外一个问题。就是重传的时候，是重传之前的一个，还是重传所有的问题。

比如对于上面的例子，是重传 Seq2 呢？还是重传 Seq2、Seq3、Seq4、Seq5 呢？因为发送端并不清楚这连续的三个 Ack 2 是谁传回来的。

根据 TCP 不同的实现，以上两种情况都是有可能的。可见，这是一把双刃剑。

为了解决不知道该重传哪些 TCP 报文，于是就有 SACK 方法。

### SACK (Selective Acknowledgement)

这种方式需要在 TCP 头部「选项」字段里加一个 SACK 的东西，它可以将缓存的地图发送给发送方，这样发送方就可以知道哪些数据收到了，哪些数据没收到，知道了这些信息，就可以只重传丢失的数据。

如下图，发送方收到了三次同样的 ACK 确认报文，于是就会触发快速重发机制，通过 SACK 信息发现只有 200~299 这段数据丢失，则重发时，就只选择了这个 TCP 段进行重复。

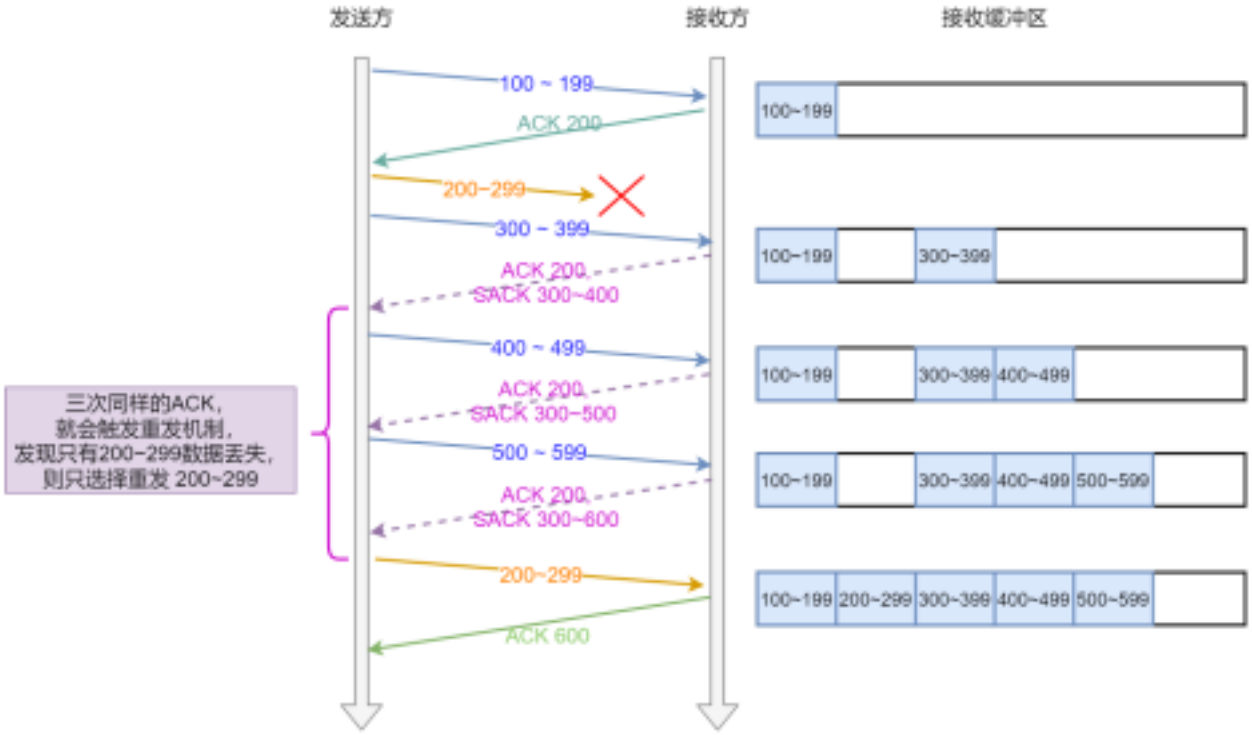



Figure 4: 如图

### D-SACK

Duplicate SACK 又称 D-SACK ，其主要使用了 SACK 来告诉「发送方」有哪些数据被重复接收了。

可见，D-SACK 有这么几个好处：

1. 可以让「发送方」知道，是发出去的包丢了，还是接收方回应的 ACK 包丢了；
2. 可以知道是不是「发送方」的数据包被网络延迟了；
3. 可以知道网络中是不是把「发送方」的数据包给复制了；

## 滑动窗口

### 引入窗口概念的原因

TCP 是每发送一个数据，都要进行一次确认应答。当上一个数据包收到了应答了，再发送下一个。

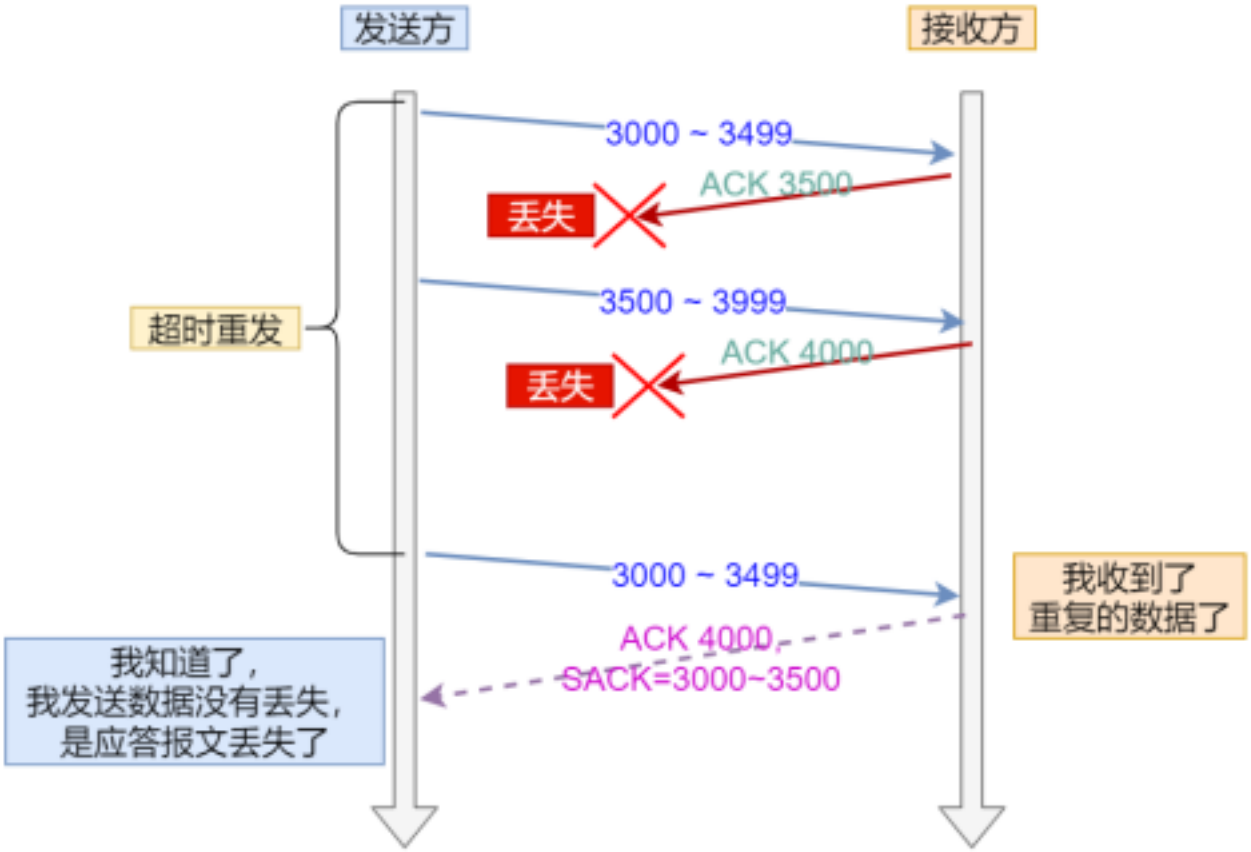
这个模式就有点像我和你面对面聊天，你一句我一句。但这种方式的缺点是效率比较低的。

这样的传输方式有一个缺点：数据包的往返时间越长，通信的效率就越低。

为解决这个问题,TCP 引入了窗口这个概念。即使在往返时间较长的情况下，它也不会降低网络通信的效率。

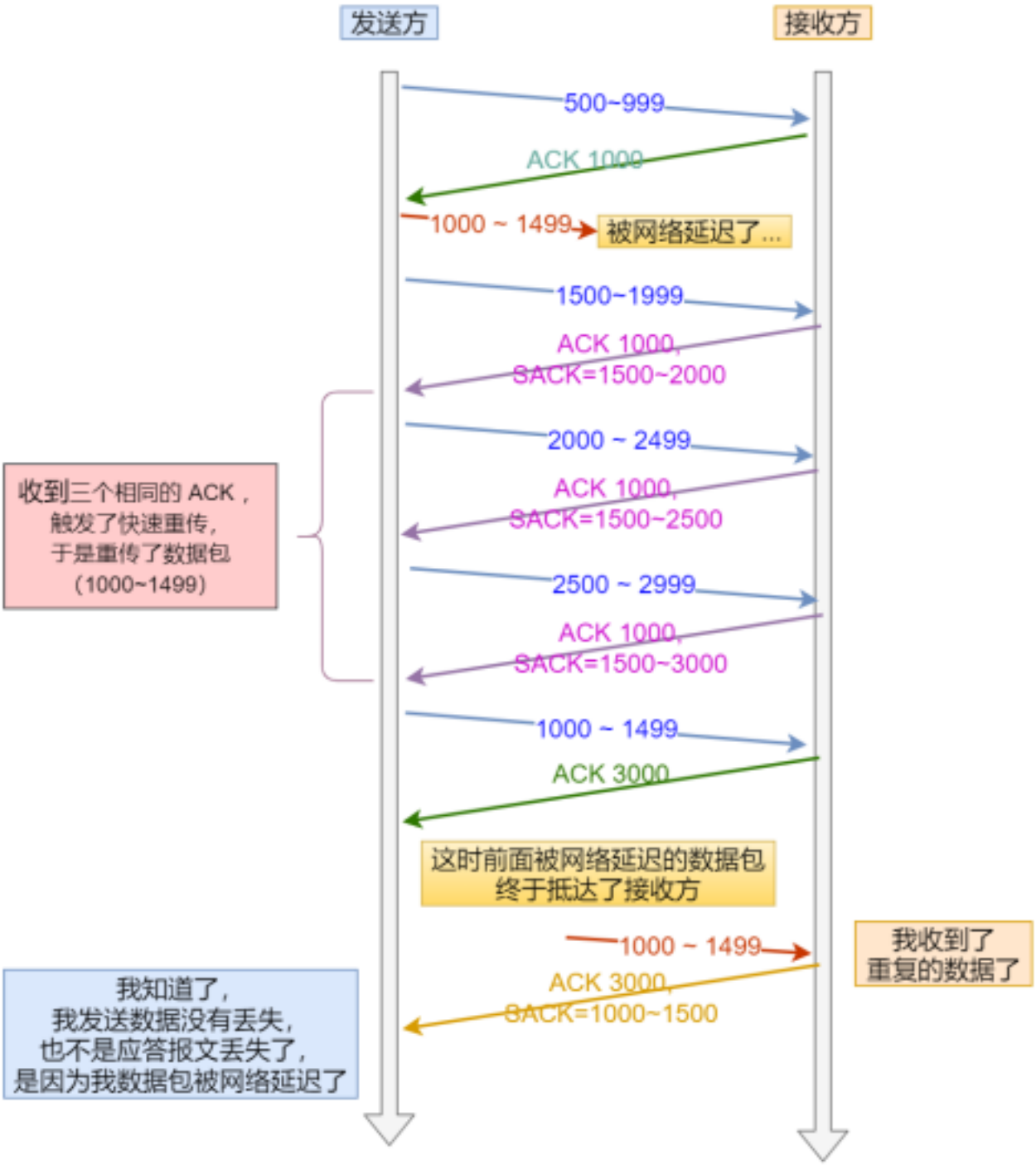
有了窗口，就可以指定窗口大小，窗口大小就是指无需等待确认应答，而可以继续发送数据的最大值。

窗口的实现实际上是操作系统开辟的一个缓存空间，发送方主机在等到确认应答返回之前，必须在缓冲区中保留已发送的数据。如果按期收到确认应答，此时数据就可以从缓存区清除。




- 「接收方」发给「发送方」的两个 ACK 确认应答都丢失了，所以发送方超时后，重传第一个数据包（3000 ~ 3499）
- **于是「接收方」发现数据是重复收到的，于是回了一个 SACK = 3000~3500**，告诉「发送方」3000~3500 的数据早已被接收了，因为 ACK 都到了 4000 了，已经意味着 4000 之前的所有数据都已收到，所以这个 SACK 就代表着 D-SACK 。
- 这样「发送方」就知道了，数据没有丢，是「接收方」的 ACK 确认报文丢了。

Figure 5: ack 丢包




- 数据包（1000~1499） 被网络延迟了，导致「发送方」没有收到 Ack 1500 的确认报文。
- 而后面报文到达的三个相同的 ACK 确认报文，就触发了快速重传机制，但是在重传后，被延迟的数据包（1000~1499）又到了「接收方」；
- **所以「接收方」回了一个 SACK=1000~1500，因为 ACK 已经到了 3000，所以这个 SACK 是 D-SACK，表示收到了重复的包。**
- 这样发送方就知道快速重传触发的原因不是发出去的包丢了，也不是因为回应的 ACK 包丢了，而是因为网络延迟了。

Figure 6: 网络延迟

假设窗口大小为 3 个 TCP 段，那么发送方就可以「连续发送」3 个 TCP 段，并且中途若有 ACK 丢失，可以通过「下一个确认应答进行确认」。如下图：

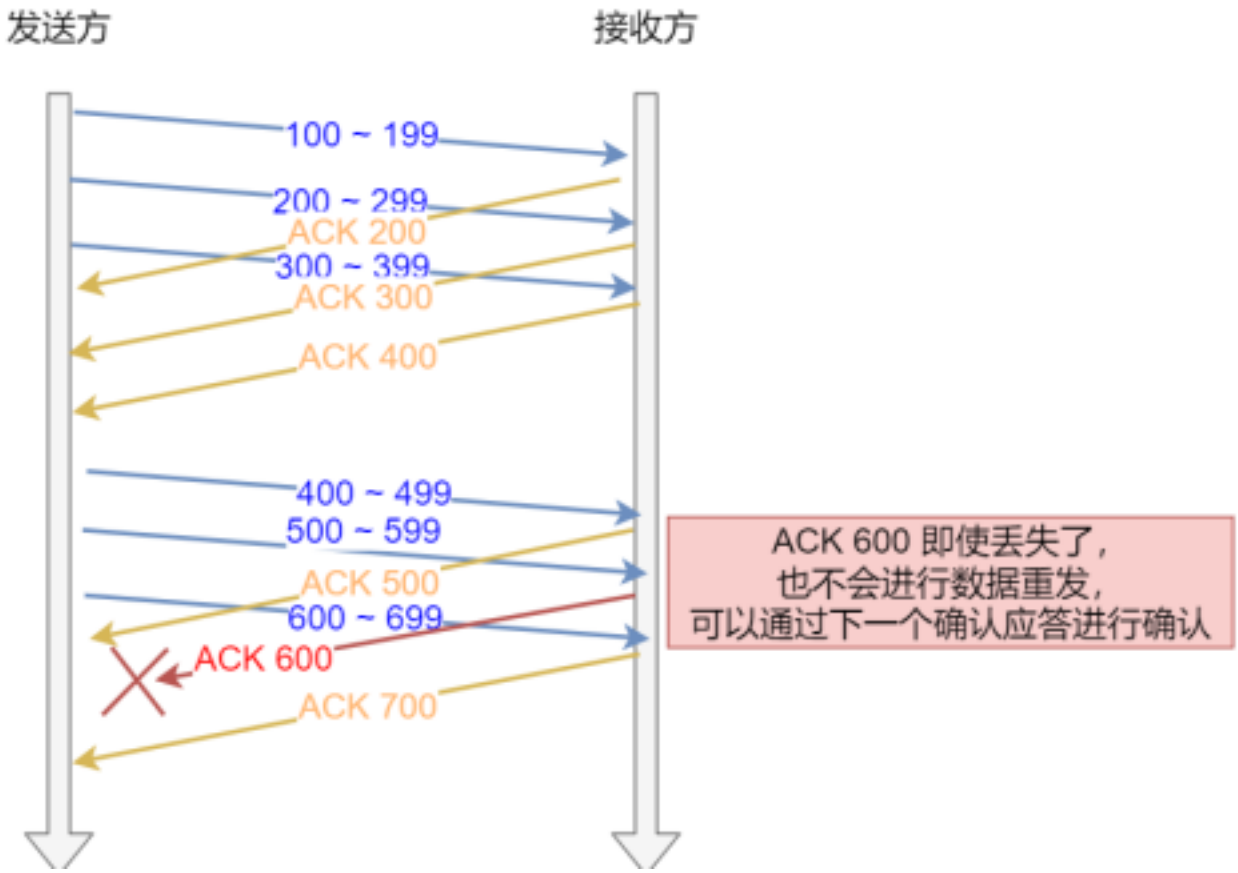



Figure 7: 示意

图中的 ACK 600 确认应答报文丢失，也没关系，因为可以通过下一个确认应答进行确认，只要发送方收到了 ACK 700 确认应答，就意味着 700 之前的所有数据「接收方」都收到了。这个模式就叫累计确认或者累计应答。

### 窗口大小由哪一方决定？

TCP 头里有一个字段叫 Window ，也就是窗口大小。

这个字段是接收端告诉发送端自己还有多少缓冲区可以接收数据。于是发送端就可以根据这个接收端的处理能力来发送数据，而不会导致接收端处理不过来。

所以，通常窗口的大小是由接收方的窗口大小来决定的。

发送方发送的数据大小不能超过接收方的窗口大小，否则接收方就无法正常接收到数据。

### 程序是如何表示发送方的四个部分的呢？

TCP 滑动窗口方案使用三个指针来跟踪在四个传输类别中的每一个类别中的字节。其中两个指针是绝对指针 (指特定的序列号)，一个是相对指针 (需要做偏移)。

SND.WND ：表示发送窗口的大小 (大小是由接收方指定的)；SND.UNA ：是一个绝对指针，它指向的是已发送但未收到确认的第一个字节的序列号，也就是 #2 的第一个字节。SND.NXT ：也是一个绝对指针，它指向未发送但可发送范围的第一个字节的序列号，也就是 #3 的第一个字节。

指向 #4 的第一个字节是个相对指针，它需要 SND.UNA 指针加上 SND.WND 大小的偏移量，就可以指向 #4 的第一个字节了。

那么可用窗口大小的计算就可以是：

可用窗口大 = SND.WND -(SND.NXT - SND.UNA)

### 接收方的滑动窗口

接下来我们看看接收方的窗口，接收窗口相对简单一些，根据处理的情况划分成三个部分：

#1 + #2 是已成功接收并确认的数据（等待应用进程读取）；

#3 是未收到数据但可以接收的数据；

#4 未收到数据并不可以接收的数据；

其中三个接收部分，使用两个指针进行划分：

RCV.WND ：表示接收窗口的大小，它会通告给发送方。

RCV.NXT ：是一个指针，它指向期望从发送方发送来的下一个数据字节的序列号，也就是 #3 的第一个字节。

指向 #4 的第一个字节是个相对指针，它需要 RCV.NXT 指针加上 RCV.WND 大小的偏移量，就可以指向 #4 的第一个字节了。

### 接收窗口和发送窗口的大小是相等的吗？

并不是完全相等，接收窗口的大小是约等于发送窗口的大小的。

因为滑动窗口并不是一成不变的。比如，当接收方的应用进程读取数据的速度非常快的话，这样的话接收窗口可以很快的就空缺出来。那么新的接收窗口大小，是通过 TCP 报文中的 Windows 字段来告诉发送方。

那么这个传输过程是存在时延的，所以接收窗口和发送窗口是约等于的关系。