网络基础 — 深入理解TCP协议

前言

前面一小节我们介绍了网络模型和一些网络协议，其中最大头TCP协议我们挖了一个坑，这一小节我们来填坑。这一小节我们主要分成三大块，第一部分我们将介绍，TCP的三次握手和四次挥手，通过简单的交互维护一个状态机之后建立一个稳定可靠的连接。我们有了一个稳定的连接，我们还需要数据包地稳定发送和接收。有了上面两个保障之后，如何充分利用网络带宽，成了最后的问题，这里我们介绍了

传统算法和BBR算法。梳理完以上这些部分，我相信你对TCP协议会有更加深入的理解，让我们先从TCP的介绍开始吧～

协议介绍

传输控制协议（TCP，Transmission Control Protocol）是为了在不可靠的互联网络上提供可靠的端到端的字节流而专门设计的传输层协议，我们前面比喻UDP是一个相信网络链路美好的单纯小朋友，那么TCP就是认识到网络链路的险恶，能保证数据的准确送达并且有各种各样问题处理稳重可靠的成功人士形象。他与UDP协议的简单不可靠相比，TCP提供较为复杂和可靠的连接服务，因此针对TCP的特点，我们可以针对性的关注以下几个方面，以及TCP的解决处理方案。

- **数据包的顺序问题**（顺序）
- **网络传输过程中的丢包问题**（丢包）
- **连接的维护问题**（连接维护）
- **流量控制问题**（流控）
- **拥塞控制问题**（拥塞控制）

协议头详解

和UDP一样，我们先看TCP的头部信息，相较于UDP头信息的简单，TCP的头部信息则复杂得多。

<table>
<tr><td colspan="3">源端口号（16位）</td><td>目标端口号（16位）</td></tr>
<tr><td colspan="4">序号（32位）</td></tr>
<tr><td colspan="4">确认序号（32位）</td></tr>
<tr><td>首部长度（4位）</td><td>保留（6位）</td><td>URG、ACK、PSH、RST、SYN、FIN</td><td>窗口大小（16位）</td></tr>
<tr><td colspan="3">校验和（16位）</td><td>紧急指针（16位）</td></tr>
<tr><td colspan="4">选项</td></tr>
<tr><td colspan="4">数据</td></tr>
</table>

这些头部信息含义分析如下：

- 源端口号：发送网络包的程序端口号。
- 目标端口号：网络包接收方的端口号。
- 序号：**发送方告诉接收方该网络包发送的数据相当于所有发送数据的第几个字节。**
- 确认序号（ACK号）：**接收方告诉发送方接收方已经收到了所有数据的第几个字节。**其中ACK是acknowleage的缩写。
- 首部长度：表示头部长度。
- 保留：该字段为保留位，现在还未使用。
- 控制位（6位）：该字段中的每个比特位分别表示以下的通信控制含义。
  - URG：表示使用紧急指针字段有效。
  - **ACK：表示接收程序数据序号字段有效，一般表示数据已经被接收方收到。**
  - PSH：表示通过flush操作发送的数据。
  - RST：**强制断开连接，用于异常中断的情况。**
  - **SYN：发送方和接收方相互确认序号，表示连接操作。**
  - **FIN：表示断开连接。**
- 窗口：**接收方告诉发送方窗口大小**（即**无需等待确认可以一起发送的数据量**）。
- 校验和：**用来检查是否出现错误。**
- 紧急指针：**表示应急处理的数据位置。**
- 选项：除了上面固定的头部字段以外，还可以添加可选字段，但除了连接操作之外，很少使用可选字段。

连接的建立与断开

我们前面提到`IP协议`是一个**无状态、无连接、不可靠**的服务，那么位于网络层的TCP协议是如何保证这个连接，我们前面提到了的这里的连接不是物理的连接，是源端和目标端维护保持的状态。通过维护连接的状态，来表示通信双方之间的连接。因此在连接建立的时候TCP要进行特殊的状态处理，连接建立的过程要经过三次通信称为三次握手，连接断开的时候要进行双方要进行四次通信，即四次挥手。

三次握手

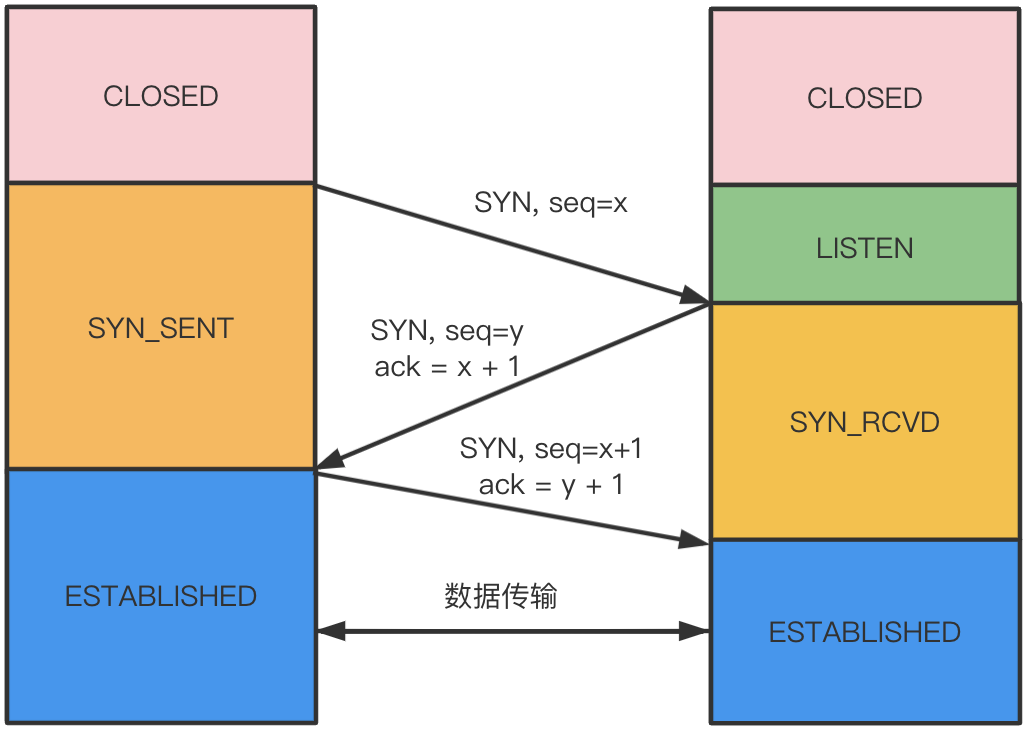
三次握手的过程并不复杂，就像两个人打招呼一样，只要双方都确认我要和对方说话，并且对方在听我说话就可以开始沟通了。就可以类比成下面的过程：

A：你好啊👋，B。

B：嗯？在听呢，你也好啊👋，A。

A：B，你好你好。🐶

双方确认对方都在听自己说话之后，两个人就可以开始愉快的沟通了。这个过程我们也常称为“请求 -> 应答 -> 应答之应答”。把这个逻辑翻译成计算机中建立网络的语言就是下面这个过程：



一开始，客户端和服务端都处于`CLOSED`状态。先是服务端主动监听某个端口，处于`LISTEN`状态。然后客户端主动发起连接`SYN`，之后处于`SYN-SENT`状态，服务端收到发起的连接，返回`SYN`，并且`ACK`客户端的`SYN`，之后处于`SYN-RCVD`状态。客户端收到服务端发送的`SYN`和`ACK`之后，发送`ACK`的`ACK`，之后处于`ESTABLISHED`状态，因为客户端已经一发一收成功了。服务端收到`ACK`的`ACK`之后，处于`ESTABLISHED`状态，此时服务端也一发一收成功了。在连接状态建立之后，双方开始进行业务数据的传输，其中在3次握手中的随后一次交互，就可以携带上层的业务数据了。在后续的的连接中如果连接空着也没关系，我们在程序中可以开启keepalive，即时没有数据传输，双方也会发送探活空包来保持通信。

> 这里有一个细节，TCP通过将每个数据包生成一个递增的序号，来解决因为网络通路延迟带来的数据包到达的顺序问题，**接收端只要收到包后按照这个序号进行排序，即可得到数据包正确的顺序**。
> 每个数据包都需要返回一个ACK包给接收端，来确认某个数据包已经到达，ACK响应也会带一个响应序号这个序号为**确认包序号+数据包长度+1**，如果发送端在一段时间内没有收到接收方响应的ACK包，发送方就会重新发送一个。**因此通过序号和ACK号，可以确认接收方是否收到了网络包。**

为什么是三次握手不是四次？双方只要一发一收（SYN+ACK）成功就可以进入到`ESTABLISHED`状态，三次交互足够让双方建立一发一收（SYN+ACK），所以再多次的“握手”可以么？可以，但是没必要浪费资源。

四次挥手

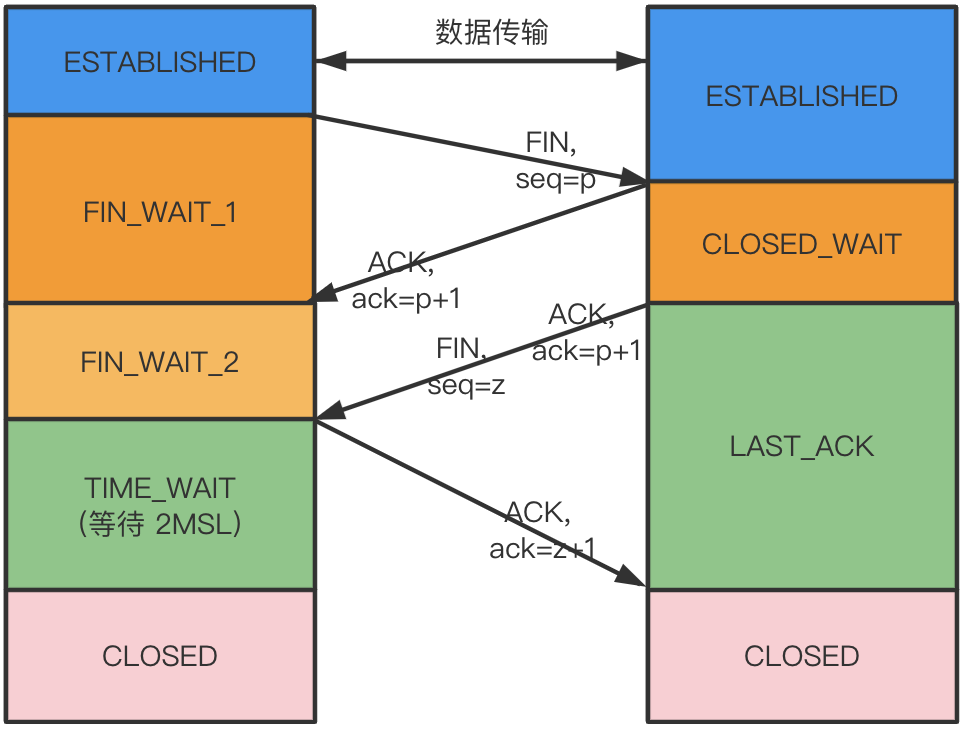
四次挥手的过程其实也很好理解，类比到我们日常生活中应该是下面这样的场景。
A：我这边结束了，再见啦👋。

B：好的，我知道了，我这边还要处理下。

B：我这边也处理完了，再会👋。

A：好的，我知道了，再见。

四次挥手和三次握手不一样，断开连接有四次交互在结束请求接收端，在第一轮通信之后，并不会立刻发送关闭请求，而是有一个`COLSED_WAIT`状态。随后再会主动发起一次关闭请求。之后请求端收到请求后`TIME_WAIT`完成之后，才会关闭连接。整个通信过程如下图：



在这个过程中，一开始双方都是处于`ESTABLISHED`的状态，随后请求端发起`FIN`请求，并随进入到`FIN_WAIT_1`状态，随后接收端收到请求后回复一个`ACK`并进入`CLOSED_WAIT`状态，请求端收到接收端的`ACK`后，进入到`FIN_WAIT_2`状态，等待接收端的发起的FIN请求。在接收端处理完一些关闭流程后，发起接收端向请求端的FIN请求，随后进入`LAST_ACK`状态，即等待最后一个`ACK`响应。请求端接收到这个`FIN`请求后，发送一个`ACK`响应并进入到时间长度为2个`MSL`的`TIME_WAIT`状态，随后关闭连接。接收端在收到请求端的最后一个`ACK`响应之后也随即关闭连接。

> 其实理解三次握手和四次挥手有个隐藏的点，注意到这个点，也就能轻松理解这个建立和关闭连接的过程了，这个点就是 TCP连接是**全双工的**，因此在建立连接阶段，需要**发送方一发一收**，确认发送端到接收端的通信正常，也需要**接收端一发一收**，确认接收端到发送端通信正常。而在四次挥手过程中，因为关闭接收方，中间会有一个`CLOSED_WAIT`状态，所以这个关闭的过程就分成了**四次通信。**
>
> 这里还有一个注意点⚠️：关闭请求端有一个`TIME_WAIT`状态，**这个状态是因为如果接收端直接关闭，这时请求端的端口又刚好被另外一个新的应用占用，新应用会收到上一个连接接收端的发出的包。**虽然seq会被重置，但是还是保险起见，发起端在发送`LAST_ACK`之后还会一个`2MSL`（MSL：

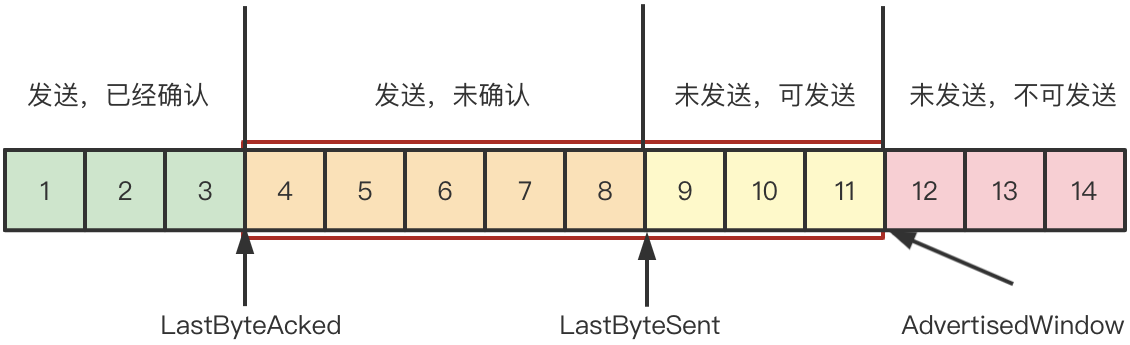
> Max Segment Lifetime，报文最大生存时间）的等待时间，然后关闭，等待这个时间是确保所有接收端发过来的包都能因为被丢弃掉。如果接收端没有收到`LAST_ACK`触发重传，这个时候接收端已经关闭了，接收端则会返回一个`RST`（强制断开连接）。其中`2MSL`，在`RFC793`中规定MSL的时间为2min，**在实际使用中，我们一般会配置为30s或者1min。**两个MSL是一个`MSL`一个是确保主动关闭方的最后ACK能够接收端，一个`MSL`是确保接收端重发的`LAST_ACK`能被请求端收到。

流量控制（滑动窗口协议）

我们前面介绍了TCP传输数据的过程中，**为了保证通信过程中不丢包，发送端每发送一个包接收端都会返回一个对应的ACK包，表示接收端接收到了这个数据包**。如果接收端和发送端之间是确认一个发送一个，那么所有的数据包发送操作都是串行的，传输效率必然很低。那么如果我们将所有的数据包，分批次一波一波发出去，这样传输效率就有了极大的提升。那问题又来了，这一波是多少数据包呢？数据包少了传输效率上不来，数据包多了接收端处理不过来，网络也容易拥塞。**TCP中通过滑动窗口协议进行流量控制。**

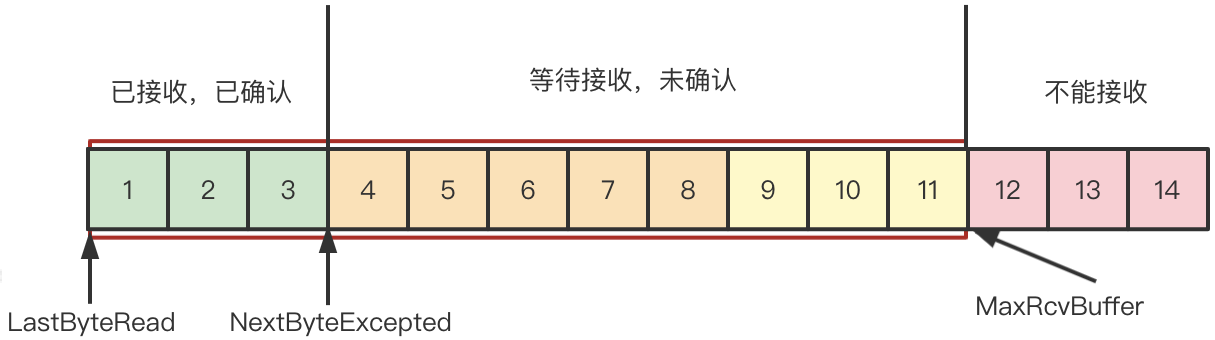
> **流量控制**解决的是**发送端速率和接收端速率不匹配**的问题，**拥塞控制**应对处理的是**网络本身的拥塞问题。**

在TCP头里面，接收端会给发送端一个窗口大小，即Advertised window。这个窗口的大小就是当前可以发送的传输窗口大小。下面就是一个发送的窗口的模拟结构，其中包含四个部分`发送已经确认`，`发送未确认`，`未发送可发送`，`未发送不可发送`。其中我们滑动窗口的范围就是所有处于**发送中**的数据。



- LastByteAcked：第一部分和第二部分的分界线。
- LastByteSent: 第二部分和第三部分的分界线。
- LastByteAcked + AdvertisedWindow：第三部分和第四部分的分界线。

对于接收端来说，也有一个对接的接收的数据结构，这个数据结构，其中包括三个部分，分别是`接收已经确认`，`等待接收未确认`，`不能接收`。

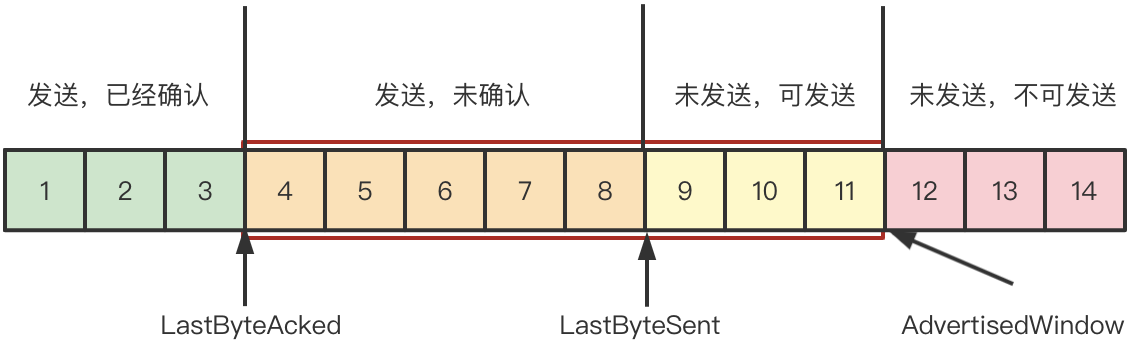


- LastByteRead 表示应用层读到的最后一个位置，这个位置后面的数据都在RcvBuffer中。
- NextByteExcepted 表示当前已接收已确认和等待接收未确认的分界线，这个分界线之前的数据是确认了且在buffer中的数据。之后是等待确认的数据。
- MaxRcvBuffer 是当前接收端的RcvBuffer的最大缓存位置。

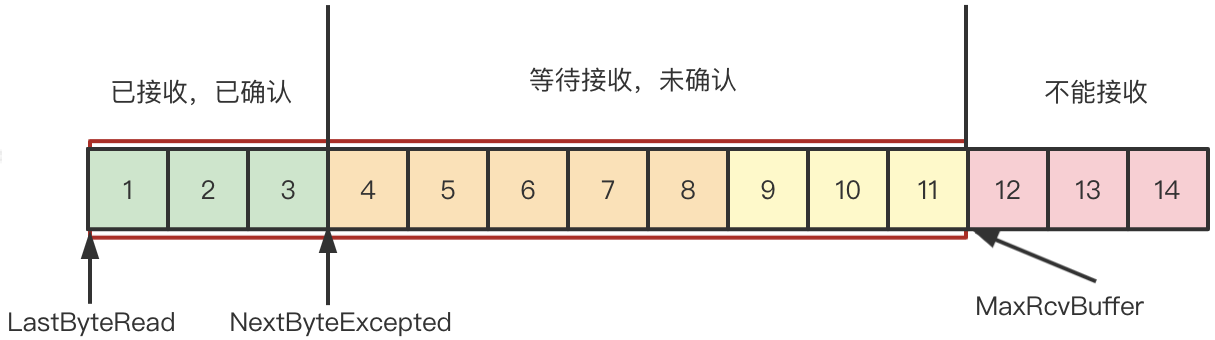
其中滑动窗口`AdvertisedWindow`的大小也就是中间等待接收但是未接收的部分，即`AdvertisedWindow = MaxRcvBuffer-NextByteExcepted`。如果当前接收端的MaxRcvBuffer在传输过程中没有发生变化的话，发送端是可以通过之前返回的窗口大小来反推当前窗口大小的。即`availableWindowSize = oldAdvertisedWindow - newLastByteAcked`。

流控分析

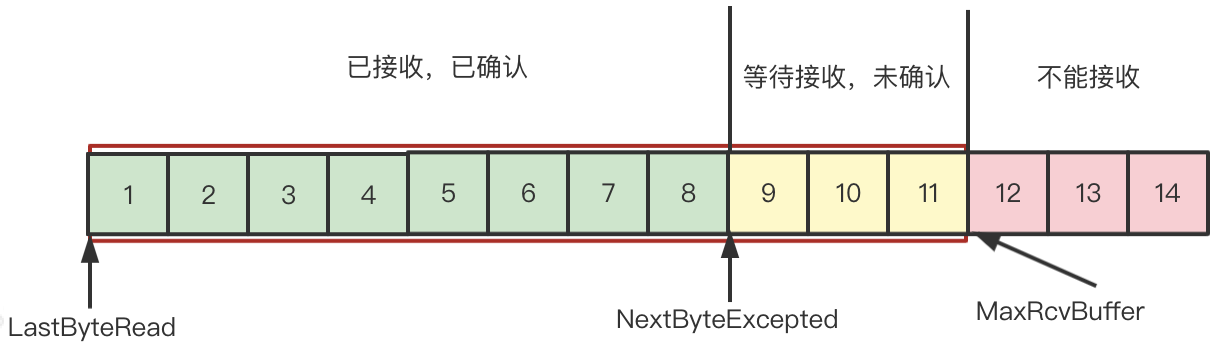
有了上面的数据结构，我们可以开始分析数据收发过程中的**流量控制和确认重试**。先看滑动窗口协议是怎么进行流量控制的。



在每次ACK的响应包中的TCP头都会携带一个窗口大小返回，这个数值就是**当前可以发送的数据包的个数**。如上图所示，当前我们的窗口大小是8，并且这次我们一次性发送了5个数据包。同时发送端的接收情况如下图：

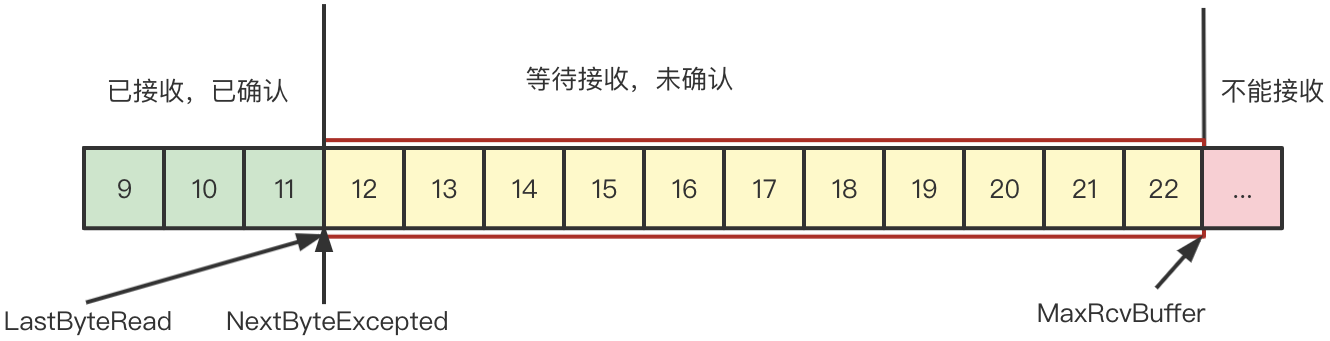


在这5个数据包确认之后，接收端的如下图所示，但是不难发现窗口并没有进行滑动。是因为数据在buffer还没有被应用层读取，此时在返回ACK包中窗口大小就8减小到3。

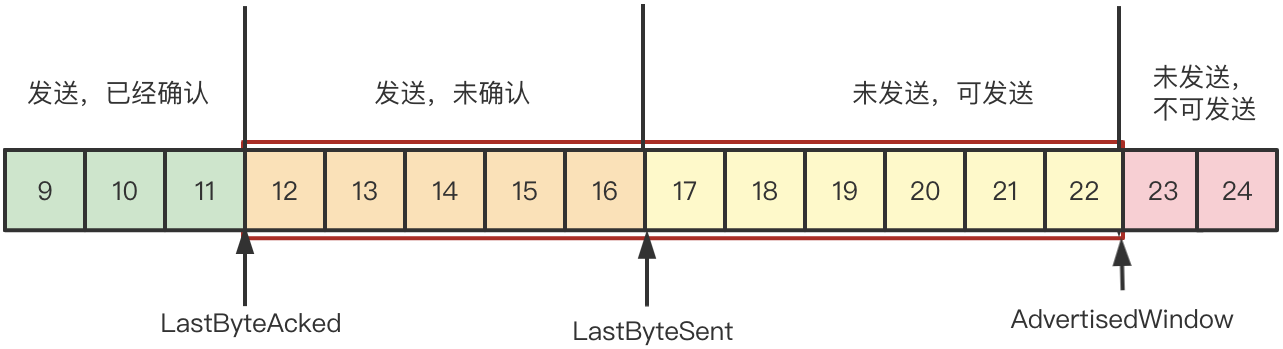


> 这里有一个注意的点⚠️
> 当窗口大小缩小到0，发送方就会停止发送数据，这个时候发送方不会给发送方发送数据包，接收方也没有多余的ACK响应告诉发送方窗口大小，这个时候双方就陷入了僵局，为了避免这种情况的发生，当窗口大小变为0之后，发送方会定时发送窗口大小探测数据包。看时候有机会调整窗口大小。但是接收方为了避免**低能窗口综合症（糊涂窗口综合症）**，接收方可以在窗口很小的时候不更新窗口，只有当接收窗口到达一定大小或者到达MaxRcvBuffer一半大小时候，才更新窗口接收数据。
>
> 上面介绍的是通过调整接收端窗口的大小处理的，还可以通过调整接收端处理即Nagle算法。
> Nagle算法，为了减少广域网中小分组数目，从而减少网络拥塞的出现，该算法要求一个TCP连接上最多只能有一个未被确认的未完成的小分组，在该分组ack到达之前不能发送其他的小分组，TCP需要收集少量的分组，并在ack到来时以一个分组的方式发出去；其中小分组的定义是小于MSS的任何分组。其中小分组就是一个数据包。
> Nagle算法本质就是动态调整发送数据包的大小，让每个数据包尽可能发送多的数据，避免发送大量过小的数据包占用浪费网络资源。
> Nagle算法和延迟ACK组合使用也会产问题。如果发送端使用Nagle算法发送一个数据包，这个时候接收端使用延迟ACK，并不会直接给发送端发送响应的ACK这个时候。发送端此时的本次要发送的长度并不满一个MSS，发送端不会立刻发送数据包，知道接收端超时发送一个ACK，发送端接收到ACK后，发送端发送数据。随后接收端和发送端会一直这样循环等待下去，从而造成较大的延迟。

而在buffer中的数据被应用层读取之后，RcvBuffer空闲出来又可以接收的数据，此时接收窗口更新并通过ACK包告诉发送端调整发送窗口大小，此时的窗口大小11，接收端如下图所示。

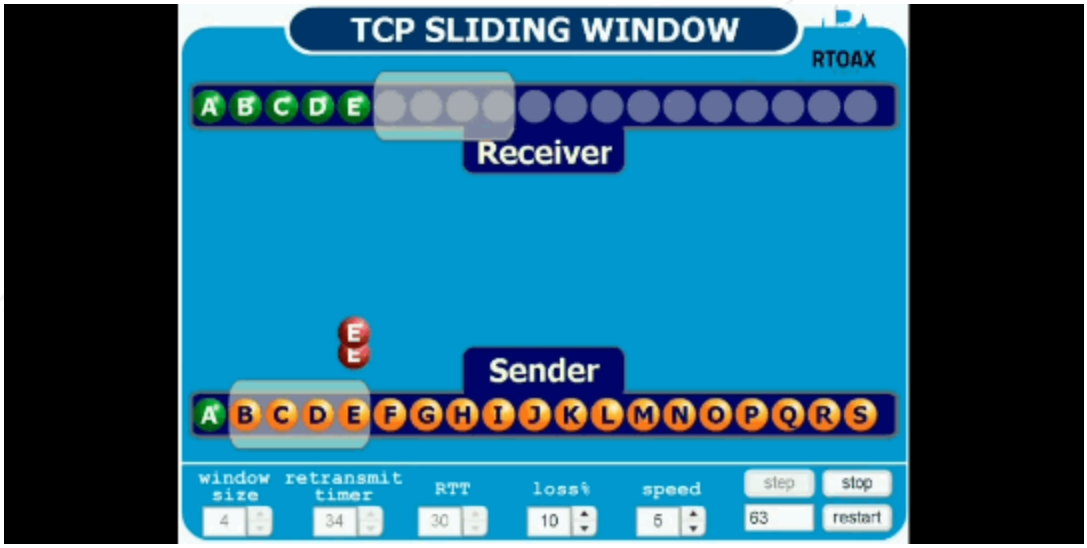


在发送端收到更新窗口大小的ACK包后，发送端也调整窗口大小为11，并继续传输数据。



> 这里要注意的是，**接收方先调整窗口大小，发送方收到接收方的ACK确认之后才会调整窗口大小。**

整个过程如下

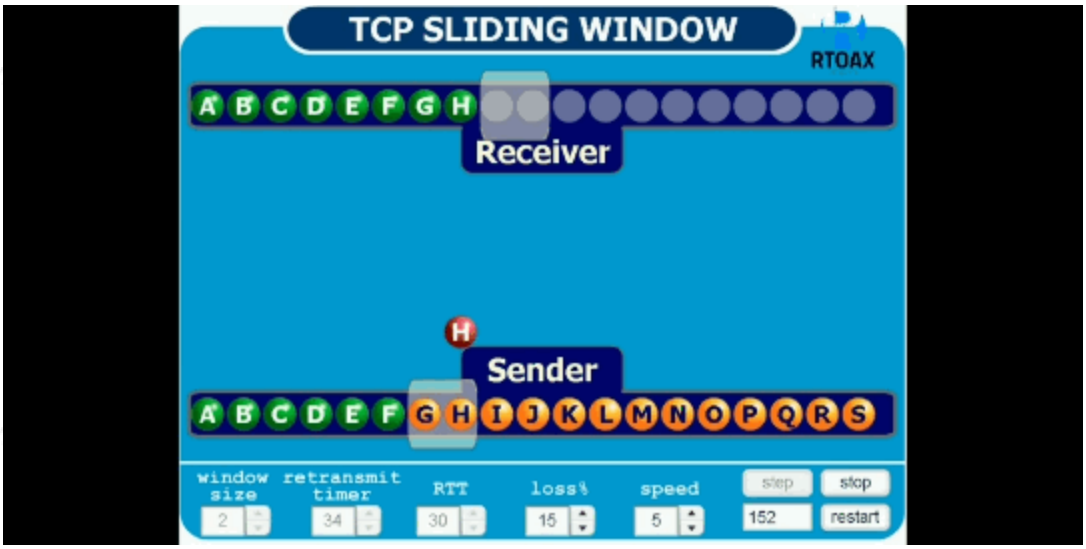


数据包确认与丢包重试

上面的过程都是一切顺利的情况，但是面对丢包时常发生的网络环境，那TCP的滑动窗口是通过什么样的数据包的确认和丢包重试机制，保证传输的效率的同时又保证了传输的可靠。

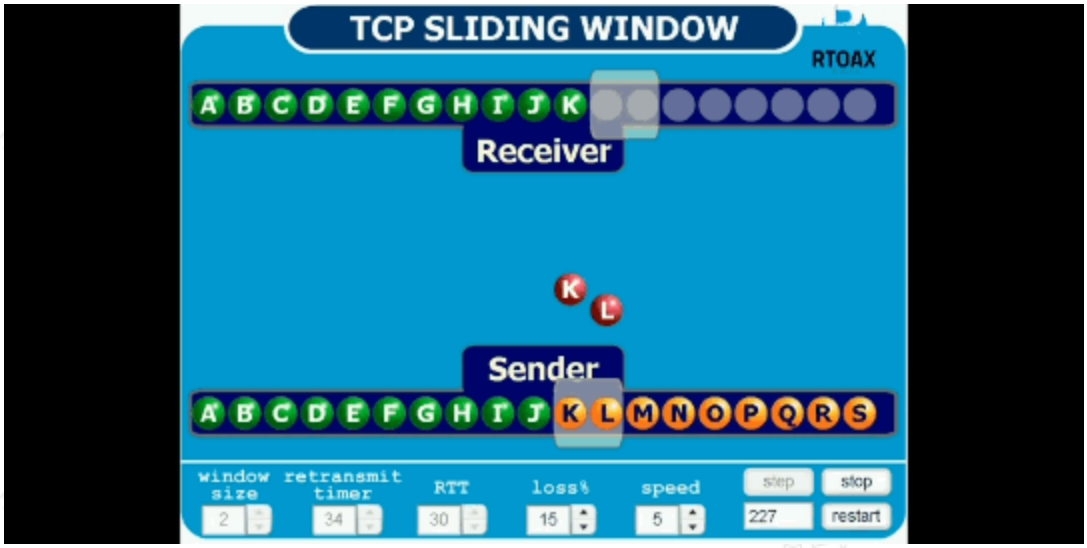
累计确认/累计应答

为了保证顺序性，每个包都有一个唯一的序列号。在建立连接的时候，就确定起始序列号的偏移量，然后按照序列号一个个发送，为了保证不丢包，对于发送的包都要进行应答，但是应答ACK也不是一个个来的，而是**只要应答所有的包的最后一个包，就表示所有的包都收到了**。这种应答模式的成为累计确认或者累计应答（cumulative acknowledgment）。这个过程类似于下面的场景，在接收端的接收到`数据包I`和`数据包J`后分别返回了`I的ACK`和`J的ACK`。`I的ACK`在传输过程中丢包了，但是发送端收到后`J的ACK`后依然确认了`数据包I`送达。



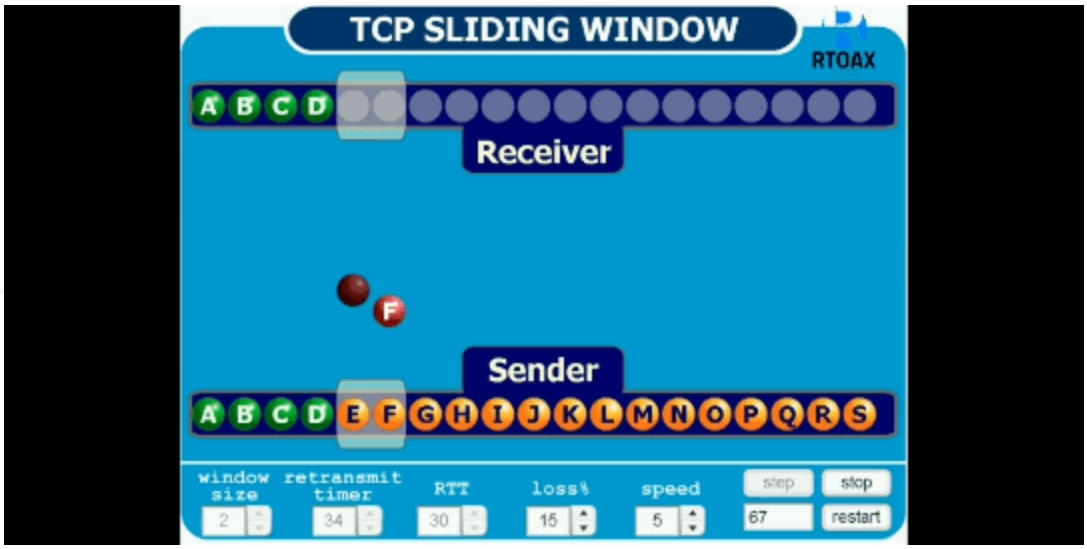
超时重传

对每一个发送的，但是没有ACK的包，都设有一个定时器，如果超过一定时间就会触发重试，但是这个超时时间不宜过长也不宜过短，时间必须大于往返时间RTT，否则会引起不必要的重传。也不易过长，否则访问时间会变慢。**这个时间需要TCP通过采样RTT时间，然后进行加权平均，算出一个均值，而且这个值还是要不断的变化，因为网络状况不断低变化**。除了采样RTT，还要采样RTT的波动范围，**计算出一个估值的超时时间**，由于重传时间是不断变化的，我们称为`自适应重传算法（Adaptive Retransmission Algorithm）`。TCP的重传策略是**超时间间隔加倍，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为设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。**



快速重传

当接收方收到一个序号大于下一个所期望的报文段时，就会监测数据流中的一个间隔，于是**它就会发送冗余的ACK，冗余ACK的期望是接收重传的报文段**。而客户端在收到**三个冗余的ACK**后，就会在定时器过期之前，重传丢失的报文段。



Selective Acknowledegment（SACK）

这种方式需要接收方在返回的ACK的TCP头中加一个SACK数据，**发送方可以通过读取SACK，知道接收方的数据接收情况，并且针对性的重传接收端丢失的包**。

拥塞控制

在上面的部分我们介绍了TCP的流量控制，通过一个数据包一个ACK的方式保证了传输的可靠性，通过各种各样的确认和重传策，数据发送端进行流量控制确保，每个数据包都能准确地传输到接收端。但是我们不仅仅只是满足于可靠的传输，我们还要充分利用网络最大带宽进行数据传输，即利用网络进行又好又快的进行传输。

> 流控控制是针对**发送接收端的发送接收策略**，而**拥塞控制是针对网络环境进行的发送流量调控策略。**

在理论中，拥塞控制一般又两种实现方式，分别是`端到端的拥塞控制`和`网络辅助拥塞控制`。其中我们TCP使用的是第一种方式。

- **端到端的拥塞控制**：**在这种拥塞控制方法中，由数据端的自己判断是否拥塞然后调整发送速率**，比如发送端的数据已经超时却还没有接收到ACK确认报文，数据往返延时过高，接收端到对同一个数据段报文重复确认等现象，我们都可以认为是网络拥塞的现象。如果发送端监测到这些现象，就应该降低数据发送的速率，如果没有，则可以慢慢提高速率。
- **网络辅助的拥塞控制**：由网络中的路由器来发送告诉发送方，网络的情况，一般有两种方式：
  - **路由器直接向发送端发送报文**，告知网络情况。
  - **路由器更改数据包中的某个标识符**，来提示网络中的拥塞情况。通常这个标识符号会被带到接收端，然后接收端再**通过ACK确认包返给发送端**。

> 网络辅助拥塞控制中的两个方法都有不太合适的地方，第一个如果通过中间设备向发送端发送反馈网络情况的数据包，不仅会增加网络环境的压力，同时也增加了发送端的数据接收压力，降低发送端吞吐。第二种方式虽然没有增加数据包的数量，但是这种方式网络拥塞的反馈延迟也更高，并且这种反馈的可达性也是一个问题。🤔

TCP需要解决的三个问题

TCP采用的`端到端的拥塞`的拥塞控制，但是从理论到实践还有很长一段路要走，其中有三个问题不得不回答。

- **TCP如何判断当前网路环境是否存在拥塞？**这个问题我们可以通过枚举的办法，枚举出丢包的场景，然后分析是否存在拥塞即可。我们可以简单分析得出一下几个场景（也就是我们前面流量控制部分重传策略中的几个场景）。
  - 若发送一条数据段后，成功接收到了接收方确认报文，则可以认为网络是没有拥塞。
  - 若发送一条数据段后，**在规定时间内没有收到确认报文（丢包或延迟太大）**，则可以认为**网络出现了拥塞。**
  - 若连续收到接收方对**同一条报文的多次冗余确认**，即可认为网络出现了拥塞（这和快速重传算机制有关）
- **TCP如何限制发送端的发送速率？**前面我们介绍了滑动窗口协议，我们可以通过控制窗口的大小来限制当前的发送速率，其中`滑动窗口RWND`通过**调整窗口大小调整发送速率，匹配接收端的接收处理速率**。还有一个`拥塞窗口CWND`，通过**调整这个窗口的大小也可以调整发送速率，来避免一次发送太多的数据包造成网络拥塞**。`LastByteSent - LastByteAcked <= min {cwnd,`

`rwnd}` **滑动窗口和拥塞窗口共同控制发送端的发送速率，发送端的发送速率必须小于或等于min(cwnd, rwnd)。**

- **TCP采用什么样的算法来控制发送速率？**即如果通过拥塞判断发送拥塞后的cwnd窗口大小调整策略，TCP调整拥塞窗口的主要算法有`慢启动`、`拥塞避免`和`快速恢复`。其中前面两个是TCP规范必须实现的，第三个则是推荐实现，TCP根据情况在这三者之间切换。

> 在介绍拥塞控制算法之前要知道一些名词，这些名词我们在后面介绍算法过程中会用到
> MSS：**最大报文段长度，TCP发送的报文段中，包含的数据部分的最大字节数。**
> cwnd：**拥塞窗口**，TCP发送但还没得到确认的报文序号都在这个区间；
> RTT：**往返时间**，发送方发送一个报文，到接收到这个报文的确认报文所经历的时间。
> ssthresh：**慢启动阈值，慢启动阶段，若cwnd的大小达到这个值，转换为拥塞避免模式。**

慢启动

慢启动是建立TCP连接后，采用的第一个调整发送速率的算法，在这个阶段，**cwnd通常会被初始化为1MSS**，这个值比较小，在这个时候，网络一般还有足够富余，而**慢启动的目的就是尽快找到上限**。在慢启动阶段，发送方每接收到一个确认报文，就会将将**cwnd的大小翻倍**，即：

- 初始的cwnd=1MSS，发送一个TCP最大报文段，成功确认后，cwnd=2MSS。
- 此时可以发送两个TCP最大报文段，成功确认后，cwnd=4MSS。
- 此时可以发送四个TCP最大报文段，成功确认后，cwnd=8MSS。
- ...

由于TCP是一次性将窗口内的所有报文发出，所以**所有报文到达并被确认的时间，近似等于一个RTT**。在这个阶段，拥塞窗口cwnd的长度在每个RTT后翻倍，发送速率也是呈指数增长。**不要被慢启动这个名字给骗了，这个过程可一点也不慢**😂 。这个阶段终究会触到一个发送的上限，当遇到以下几种情况时候，cwnd 将进行调整以适应当前网络。

1. 在慢启动的过程中，发生了**数据传输超时**，则此时**TCP将ssthresh的值设置为**`cwnd/2` ，然后将**cwnd重新设置为1MSS**，**重新开始慢启动**过程，这个过程可以理解为试探上限。
2. 第一步试探出的上限ssthresh将用在此处。**若cwnd的值增加到 >= ssthresh时**，此时若继续使用慢启动的翻倍增长方式，必然很快达到速率上限出现网络拥塞。所以也就是这个时候**慢启动结束，改为拥塞避免模式。**
3. 若发送方接收到某个报文的**三次冗余确认（触发了快速重传条件）**，则**进入到快速恢复阶段，同时ssthresh = cwnd / 2**，毕竟发生快速重传也可以认为是发生拥塞导致的丢包，此时可以设置 **cwd = ssthresh + 3MSS**。

以上就是慢启动算法（模式）的一些细节和处理点。

拥塞避免

在慢启动阶段，当`cwnd >= ssthresh`时候，为了避免很快接近拥塞阈值，慢启动结束拥塞控制启动。拥塞避免阶段是一个速率慢且线性增长的过程，在这个模式下，**每经历一个RTT，cwnd的大小增加**

1MSS。这个线性增长什么时候结束呢？分以下两种情况。

1. 在这个过程中**发生了超时，则表示网络拥塞**，这个时候ssthresh被修改成`cwnd/2`，然后cwnd被只为1MSS，并且进入慢启动阶段。
2. 若发送方接收到了**某个报文的三次冗余确认**（即触发了快速重传的条件），此时也认为发生了拥塞，则ssthresh 被修改为 cwnd/2，然后cwnd被设置为**ssthresh + 3MSS**，并进入快速恢复阶段。

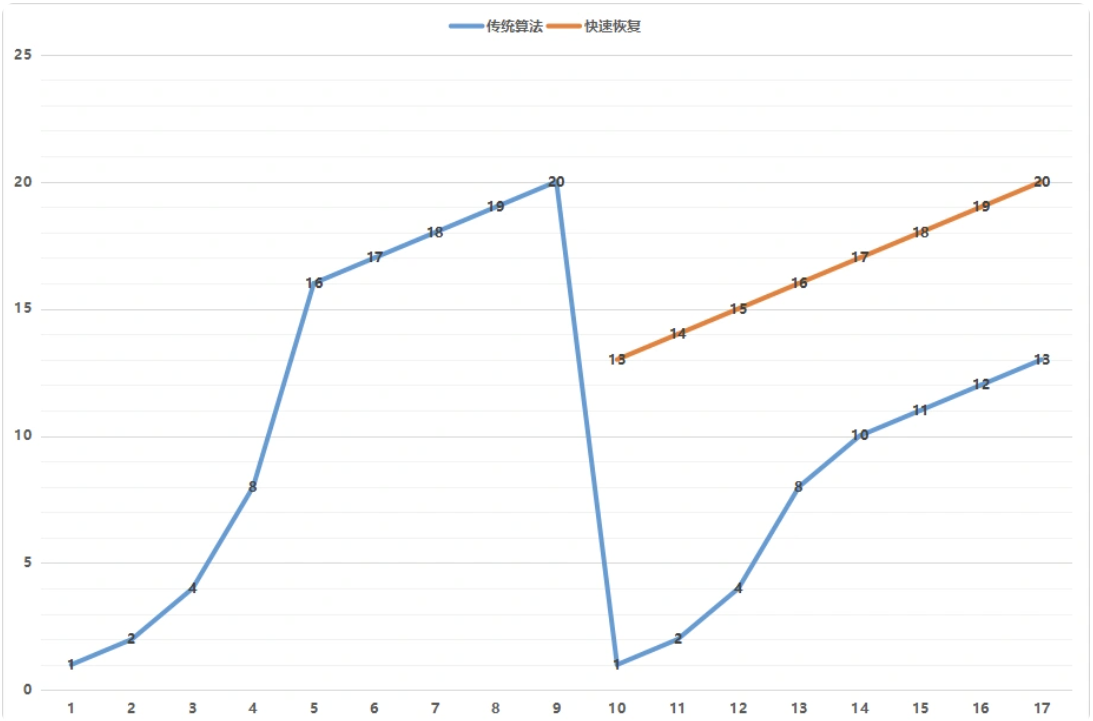
拥塞避免阶段是在慢开始之后提升发送速率的阶段，让发送速率尽可能保持一个较高的水平上。也可以理解为避免拥塞的缓慢提神发送速率阶段。

快速恢复

快速恢复模式和上面两种模式不太一样，这种模式在TCP规范中**没有要求强制实现**，只是一种**推荐实现**的模式。在快速恢复阶段，**每接收到一个冗余的确认报文，cwnd就增加一个MSS**，其余不变。在发生以下两种情况，退出快速恢复模式。

- 在快速恢复过程中，**计时器超时**，这时候，ssthresh被修改为`cwnd/2`，然后cwnd被设置为1MSS，并进入**慢启动**模式。
- 若发送方接收到**一条新的确认报文**，则cwnd被置为ssthresh，然后进入到**拥塞避免**模式。

这几个算法传输速率的图如下，其中蓝色的传统算法的第一段，0到16为慢启动阶段，16到20这个阶段以1MSS/RTT增速提升cwnd大小，橙色的线代表快速恢复算法，可以看到在10这个点如果出现了丢包则会进入下面蓝色的快启动阶段，如果出现的是冗余确认，进入的是快速恢复的橙色线条，这个时启始cwnd = (20/2) + 3 = 13，并随后进入拥塞避免阶段。

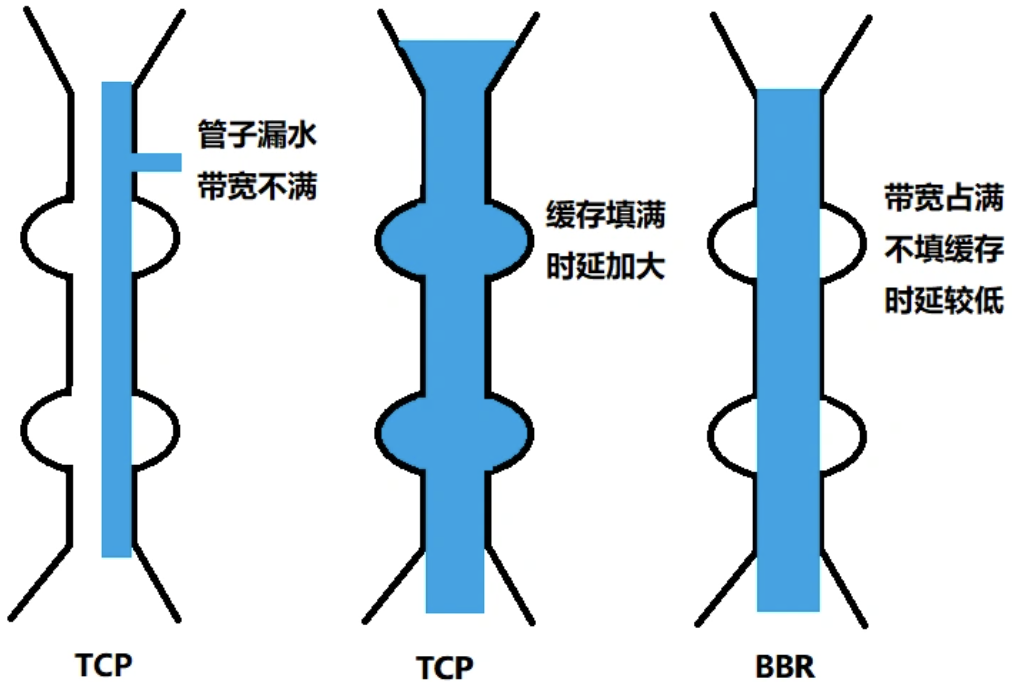


BBR算法介绍

上面的算法通过接收端丢包和冗余ACK包来判断当前网络的拥塞情况，然后调整自己的发送速率，因此有了拥塞控制算法，慢启动、拥塞避免和快速恢复算法（模式）。但是这样的“知进退”的算法，在某些延迟很重要的场景下却降低了传输速率。TCP判断拥塞的两个条件本身也是不够严谨的。

- **第一个问题是丢包不一定就代表网络通道满了**，可能整个通道就是漏水的。有时候公网上即时速率跑不满，也会发生丢包的场景，这并不代表发生拥塞了。
- **第二个问题是TCP的拥塞算法要等中间设备全部都填满了，才会发生丢包**。这个时候降低速率已经为时已晚了。其实传输的过程中只要跑慢网络传输线路就可以了。

为了优化这两个问题，后来有了 TCP BBR 拥塞算法。它企图找到一个**平衡点**，通过不断地加快发送速度，将管道填满，但是不要填满中间中间设备的缓存。这个平衡点可以很好的达到高带宽和低延迟的平衡。



> 我们这里只是入门介绍BBR算法解决的问题，感兴趣的同学可以阅读BBR算法有关论文。
> https://queue.acm.org/detail.cfm?id=3022184 😏

总结

这一小节我们详细总结了TCP协议的细节，从TCP的协议头开始，详细介绍了了协议头里面的一些标识，通过这些标识的维护，实现TCP的连接建立与断开，流量控制和拥塞控制等功能。随后我们介绍了TCP可靠连接的建立和断开，以及三次握手和四次握手的交互细节。在建立连接之后我们开始传输数据进行流量控制。TCP使用滑动窗口的接口作为流控的基础。TCP的发送方都有一个窗口，其中发送方的窗口依赖接收方窗口大小反馈。 通过这个窗口大小的的调整，发送端调整发送端发送速率，解决发送端

发送速率和接收端发送速率不匹配的问题。随后我们介绍了滑动窗口的确认和应答机制，其中有**累计确认、超时重传和快速重传**等机制，确保了TCP传输的可靠性和数据包的顺序性。最后在有了以上的基础，我们介绍了针对网络环境的TCP拥塞控制，其中介绍了传统算法的必须实现的慢启动、拥塞避免和可选实现的快速恢复。通过这两小节的学习，我们建立一个以开发中接触较多的协议为核心的网络基础。接下来我们将进入网络编程的学习，加油～

学习资料

- 计算机网络–TCP的拥塞控制(超详细)
- 极客时间专栏《趣谈网络协（第八讲）》
- 极客时间专栏《趣谈网络协（第十一讲）》
- 极客时间专栏《趣谈网络协（第十二讲）》