TCP拥塞控制之1： 拥塞控制(congestion control)

Notebook:	Net		
Created:	2020/2/4 11:41	**Updated:**	2020/2/5 15:34
Author:	kursk.ye@gmail.com		
URL:	https://en.m.wikipedia.org/wiki/TCP_congestion_control		

最近研究了一下TCP的拥塞控制，感觉收获很大，本文大部分内容来自wiki ,对wiki的翻译和整理。

Transmission Control Protocol (TCP) uses a network congestion-avoidance algorithm that includes various aspects of an additive increase/multiplicative decrease (AIMD) scheme, along with other schemes including **slow start** and **congestion window**, to achieve congestion avoidance. The **TCP congestion-avoidance algorithm** is the primary basis for congestion control in the Internet.[1][2][3][4] Per the end-to-end principle, congestion control is largely a function of internet hosts, not the network itself. There are several variations and versions of the algorithm implemented in protocol stacks of operating systems of computers that connect to the Internet.

TCP通过网络拥塞规避算法(**TCP congestion-avoidance algorithm**)，该算法包括了各种不同观地AIMD方案，另外还结合慢开始(slow start)和拥塞窗口(congestion window)机制。TCP拥塞规避算法是互联网拥塞控制的基础，在端到端的传递原则中，拥塞控制由互联网中的主机负责，而不是网络本身，有很多种版本的拥塞算法实现，它们广泛存在于联接在互联网上电脑的操作系统的协议栈中。

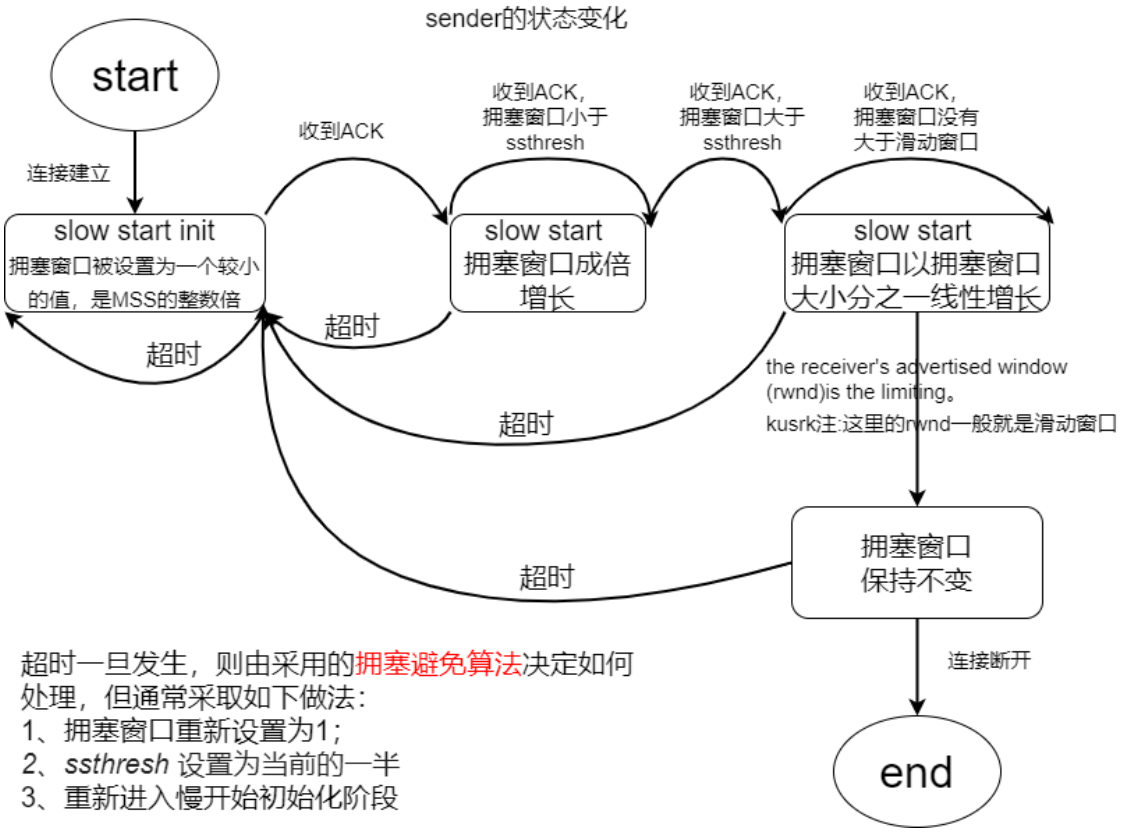
其实现过程如下：

To avoid congestive collapse, TCP uses a multi-faceted congestion-control strategy. For each connection, TCP maintains a congestion window, limiting the total number of unacknowledged packets that may be in transit end-to-end. This is somewhat analogous to TCP's sliding window used for flow control. TCP uses a mechanism called slow start[1] to increase the congestion window after a connection is initialized or after a timeout. It starts with a window, a small multiple of the maximum segment size (MSS) in size. Although the initial rate is low, the rate of increase is very rapid; for every packet acknowledged, the congestion window increases by 1 MSS so that the congestion window effectively doubles for every round-trip time (RTT).

如果网络长时间承担传输超过其本身能力限制的传输数据量，就会导致其拥塞崩溃(Congestive collapse)，TCP使用多因素(multi-faceted)拥塞控制策略，对于每一次联接，TCP都产生一个拥塞窗口(congestion window)，这个拥塞窗口限制了还没有被确认(unacknowledged)、下一步准备要被传输的包的总数。这一思想仿造了TCP的滑动窗口的流控制思路.TCP使用慢开始机制，在每次连接初始化或者超时以后增加拥塞窗口，拥塞窗口开始时是一个比较小的值，但必须是MSS(maximum segment size)的整数倍，虽然开始很小，但增长率却很快，每次收到ACK，拥塞窗口就会乘倍增长(kursk注：这里的英文我认为有问题，不是increase by 1 MSS，wiki在对拥塞窗口的详细解释中于本处有不同，详细解释中更清楚)

我自己的理解如下

拥塞控制是由发送者(sender也有wiki文字写为trasmitter)负责的,发送者的状态变化如下图



当连接建立后，发送者首先进入慢开始初始化阶段，此是拥塞窗口可能为1、2、4或10，总之是一个比较小的值，但是为MSS的整数倍(参考)。
MSS可以理解为package中每个段的大小(参考)

然后，发送者通过AIMD机制的算法来决定拥塞窗口是变大，还是变小。
AIMD是一种反馈算法，它通过一个反馈因素(一般是超时或者丢包)来判断算法的结果，如下图就是及其简单的AIMD算法

$$w(t+1) = \begin{cases} w(t) + a & \text{if congestion is not detected} \\ w(t) \times b & \text{if congestion is detected} \end{cases}$$

如果发送者收到ACK，说明接收者接收到了包，那么则拥塞窗口*2，即成倍增长(参考 Slow start begins initially with a congestion window size (CWND) of 1, 2, 4 or 10 MSS.[5][3]:1 The value for the congestion window size will be increased by one with each acknowledgement (ACK) received, effectively doubling the window size each round-trip time)；

如果发送者超时时间内没有收到ACK，要么是网路问题，要么是接收者出现问题，这种情况会当作网络问题处理，根据采用的拥塞算法来决定如何处理，比如常见的 TCP Tahoe、 TCP Reno算法等等，可参考wiki ,当前我正在学习的bbr算法就属于其中之一。

但是，超时发生时发送者一般情况下会做三件事:
1、将用拥塞窗口重新设置为1
2、将ssthresh设置为当前ssthresh的一半大小
3、重新进入慢开始初始化状态

ssthresh的全称是slow-start threshold，即慢开始拥塞窗口的阀值，linux下这个值初始值设置得比较大，但是在过程中会调节。

如果一切正常，拥塞窗口一直在增长，那么就有如下几种可能
一是拥塞窗口超过ssthresh大小，则拥塞窗口会以当前拥塞窗口大小的分之一线性增长，只到发生超时。(the congestion window increases linearly at the rate of 1/(congestion window) segment on each new acknowledgement received. The window keeps growing until a timeout occurs.)
二是达到了接收方的宣告值(the receiver's advertised window (rwnd) is the limiting facto),这里的rwnd一般就是滑动窗口的值(Sliding window),则拥塞窗口停止增长。

滑动窗口协议(Sliding window protocol)是基于包(packge)的传输协议，是一种按序发送的可信传输协议。(参考)wiki中解释得已经相当清楚，可以参考youtube的这个视频，滑动窗口是一种很重要的设计思想，不仅用在2层和4层中，像kafka这种分布式消息中间件，也使用了滑动窗口的思想。

注意滑动窗口的几个关键点:

1、**滑动窗口解决什么问题**。之所以设计出滑动窗口这个机制，是为了避免每次发送者发送一次后，只能等收到这次发送的ACK，才能进行下次发送，导致连接中等待时间超过了发送时间，没有有效利用网络资源。通过滑动窗口这个设计，由接收者宣告自己能够接收的窗口大小，发送者根据这个窗口就可以自己决定一些字节能否发送——无论是否收到已经发送的字节的ACK，发送者的字节只要在这个窗口内，就可以发送，避免了将连接中长时间的等待出现。
2、**滑动窗口何时滑动**。发送者只有收到ACK了，才将发送者的滑动窗口向后滑动。
3、**滑动窗口由谁控制**。滑动窗口与由发送者控制拥塞窗口最显著的不同点在于，滑动窗口由接收者控制，TCP中有16个字节用于存放滑动窗口的大小，发送者通过在TCP header存放滑动窗口的值，告知发送者应该把自己的滑动窗口设置为多大，从而保证发送者和接收者的滑动窗口大小一致。

回到拥塞控制的问题，既然滑动窗口机制让发送者决定是否发送时，不再依赖于是否收到已发送字节的ACK，那么怎么解决重发问题。

根据wiki的解释，一般情况下，发送者自己有一个定时器，每次发送后启动这个定时器，如果在一个特定时间内，还没有收到ACK，就判断该次发送失败，会重新发送。
但是这样效率太低了，快速重发(Fast retransmit)因此被设计出来。

重复ACK请求是快速重发机制的基础，重复ACK请求是指发送者多次收到接收者对于同一个ACK请求，而之所以接收者会重复请求同一个ACK，是因为滑动窗口(或者说拥塞窗口)让发送者可以一次性发出多个有序的字节。如下图:

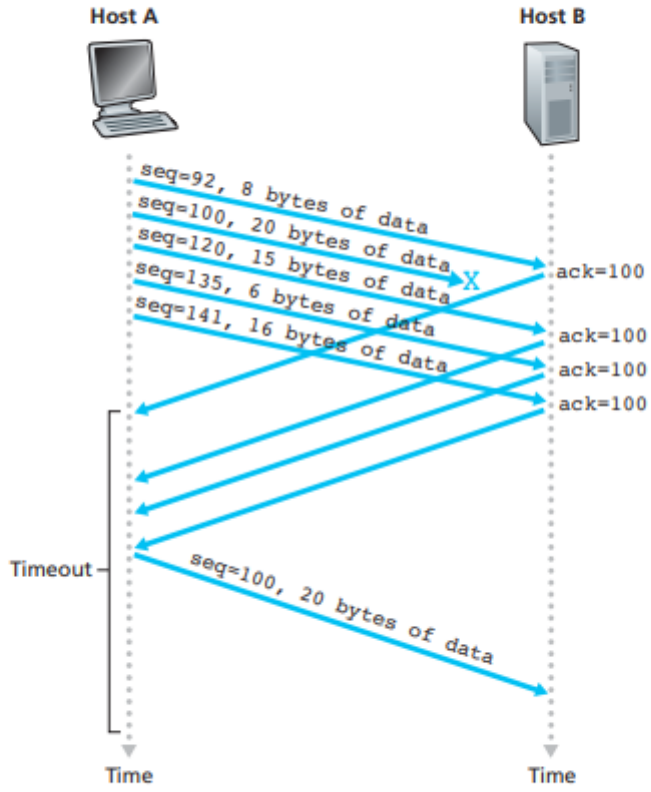


Figure 3.37 ♦ Fast retransmit: retransmitting the missing segment before the segment's timer expires

host A重新发送seq=100的segment时，是在超时时间内，那为什么超时时间未到，发送者就重发呢？因为Host B收到seq=92,120,135,141的segment,按segment应该是有序的，所以接收者知道seq=100的segment肯定丢失了，所以在在收到seq=120,135,141的segment后，每次都向发送者要求重发seq=100的segment，快速重发机制明确，发送者三次收到对同一个ACK的请求，就认为该segment丢失，不等待超时时间，直接重发(When a sender receives three duplicate acknowledgements, it can be reasonably confident that the segment carrying the data that followed the last in order byte specified in the acknowledgment was lost. A sender with fast retransmit will then retransmit this packet immediately without waiting for its timeout.)

快速重发让重发不再依赖超时事件，通过增加重发的频率，从而提高通讯的效率。