现代互联网的TCP拥塞控制(CC)算法评谈

原创

dog250 于 2018-08-25 08:53:00 发布 9293 收藏 14
文章标签： TCP 拥塞控制

本文链接：https://blog.csdn.net/dog250/article/details/81973915

动机

写这篇文章本质上的动机是因为前天发了一个朋友圈，见最后的***写在最后***，但实际上，我早就想总结总结TCP拥塞控制算法点点滴滴了，上周总结了一张图，这周接着那些，写点文字。

前些天，Linux中国微信公众号推了一篇文章，上班路上仔细阅读了一下，感受颇深，一方面由于此文说出了很多我想表达却苦于组织语言的观点，另一方面，此文表达了一些我没能认识到的事实，整个一天的时间，我都在思考此文的字词句，我想写一篇读后感是最合适不过的了。

附上那篇文章的链接：
《netdev 第二天：从网络代码中移除“尽可能快”这个目标》：https://linux.cn/article-9935-1.html
由于此文是译文，所以我把原文也附上：
《netdev day 2: moving away from “as fast as possible” in networking code》：https://jvns.ca/blog/2018/07/12/netdev-day-2–moving-away-from–as-fast-as-possible/
以下当我说《第二天》的时候，说的就是这篇文章。

OK，且听我继续唠叨。

要聊TCP拥塞控制，如果你没有读过Van Jacobson(即大名鼎鼎的范.雅各布森)的那篇讲TCP拥塞控制的经典论文，那就根本不好思议说自己是混这个行当的，这篇论文的链接我也附上：
《Congestion Avoidance and Control》：http://ee.lbl.gov/papers/congavoid.pdf
当然了，你可以以任何其它途径搜索到这篇论文。

本文并非针对这篇论文的评谈，之所以在文章的开头我就提到这篇论文，是因为《第二天》在开头也引用了这篇论文，这篇论文非常重要，因为它道出了拥塞控制的本质问题和根本的解法，在我评谈完《第二天》后，我会以大量的篇幅去解释这篇论文。

首先，我先从《第二天》开始。

关于《netdev 第二天：从网络代码中移除“尽可能快”这个目标》

《第二天》讲述了一个事实，即***互联网从1988年到2018年这30年间发生了变化***，该变化要求我们的TCP拥塞控制算法***必须***做出改变，否则，***互联网将崩塌***！

这不是危言耸听，这是真的。让我们来回顾一段历史。

还记得1988年Van Jacobson那篇经典论文的写作背景吗？嗯，是的，1986年那时互联网经历了一次崩塌！嗯，1988年是一个分水岭：

- **1988年之前**：TCP毫无拥塞控制
  简单的端到端滑动窗口流控机制，一切都都工作的很好！直到1988年前夕的1986年网络崩溃。
- **1988年之后**：引入了TCP拥塞控制
  引入慢启动，拥塞避免这两个概念，此后TCP拥塞控制将在这两个基本概念的基础之上展开进化，直到…下一个分水岭，嗯，2016年Google放出BBR(事实上Google早就在内部预研并实施BBR了)。

很多人可能并不知道这个事实，但事实真的是，1988年以前，TCP是没有什么拥塞控制算法的，能发多少数据包全凭对端通告过来的那个TCP协议头里的***窗口***来决定，自从1974年TCP协议诞生，经历1983年的标准化一直到1988年前夕，这种机制一直工作的非常好，那么到底是什么原因导致了1988年前夕的网络崩溃进而导致拥塞控制被引入TCP呢？

这一切的答案藏匿于时间！

我们用发展的眼光来看待这段历史。非常显然，1988年接入网络的TCP终端数量要远远大于1974年或者1983年，这背后的推动力来自于网络本身规模的指数级增长。我们看一下网络节点的变化情况，由于本文不是专门讲网络本身的，所以为了一种延续性，我将早期的网络统称为ARPA网，虽然它的名字一直在变…

- 1969年： ARPAnet初创，一共4个节点
- 1970年：13个节点构成ARPA的全部
- **1972年**：法国CYCLADES网络！！！(**非常关键**)
- 1973年：发展到约40个节点
- **1974年5月**：《分组网络信息交换协议》发布，TCP协议诞生
- **1974年12月**：RFC675发布，互联网标准诞生
- 1976年：持续增加节点到57个
- 1982年：网络节点增长到100个左右
- **1983年1月1日**：TCP/IP取代旧协议族成为ARPA网控制协议
- **1984年**：美国国防部***将TCP/IP作为计算机网络的标准***
- 1984年：节点数量达到1000+，已经初具规模，互联网雏形形成
- ***1986年：网络拥塞导致崩溃***
- 1988年：拥塞控制被引入TCP

这期间，随着TCP协议被标准化，TCP几乎成了默认的接入协议…

这就像中国的汽车产业一样，80年代我们中国也许只有机关单位拥有少量的汽车，随着时间的推移，汽车变得越来越多，还记得小学的课文《北京立交桥》吗？在20世纪90年代，立体交通被首先引入北京，上海，广州等城市，这个和互联网的发展如出一辙，所谓日光之下，并无新事！

非常简单的道理，节点多了，必然会拥塞。然而这是导致拥塞的唯一原因吗？

我们上大学第一节计算机网络课的时候，老师一般会介绍***网络可以分为通信子网和资源子网***。上述的ARPAnet发展脉络说的更多的是通信子网，到底何谓通信子网？

请注意上面发展脉络注解中的***1972年法国CYCLADES网络***，它首次提出了***端到端原则***，即：

> The CYCLADES network was the first to make the hosts responsible for the reliable delivery of data, rather than this being a centralized service of the network itself. Datagrams were exchanged on the network using transport protocols that do not guarantee reliable delivery, but only attempt best-effort. To empower the network leaves, the hosts, to perform error-correction, the network ensured end-to-end protocol transparency, a concept later to be known as the end-to-end principle .

上述引用来自CYCLADES的Wiki页面：https://en.wikipedia.org/wiki/CYCLADES
这个端到端原则非常重要！后来在1974年出炉的TCP协议就是一个非常成功的端到端协议，它保证了网络对资源的透明性，我们现在知道，TCP几乎不了解网络的任何细节，这一切思想全部来源于那场端到端原则引发的头脑风暴！

比较悲哀，先行者往往超越了时代，最终CYCLADES消亡了，一方面它太超前，另一方面可能是欧洲实在没有类似美国的这种互联网基因，它最终退出了历史，关于CYCLADES，下面这个链接不错：
https://www.techopedia.com/definition/27855/cyclades

花开两朵，各表一枝。我们再看看同时期的另外一条线，即资源子网的发展脉络。所谓的资源子网，其实更多的指的就是计算机终端。

- 1972年：Intel 8008处理器发布
- 1974年：Intel 8080处理器发布
- 1976年：《以太网：区域计算机网络的分布式数据包交换技术》发布，以太网诞生
- 1978年：Intel 8086处理器发布
- 1980年：微软dos操作系统
- 1985年：Intel 80386处理器发布

时间告诉我们一个事实，处理器越来越强大，计算机终端在进步！这背后是一个摩尔定律在悄悄发挥着作用！

这意味着什么？这意味着承载TCP的计算机终端，将会以越来越快的速度发送数据包！

节点越来越多，发包越来越快，这便是1988年前夕网络拥塞的根源！

于是，1988年后，拥塞控制被引入了TCP协议，这便是我们再熟悉不过的现代TCP协议了，如果你手头有Linux内核源码，看看net/ipv4/tcp_cong.c这个文件，tcp_reno_cong_avoid函数，就是描述的Van Jacobson论文里的思想：

```
/*
 * TCP Reno congestion control
 * This is special case used for fallback as well.
 */
/* This is Jacobson's slow start and congestion avoidance.
 * SIGCOMM '88, p. 328.
 */
void tcp_reno_cong_avoid(struct sock *sk, u32 ack, u32 acked)
{
    struct tcp_sock *tp = tcp_sk(sk);

    if (!tcp_is_cwnd_limited(sk))
        return;

    /* In "safe" area, increase. */
    if (tcp_in_slow_start(tp)) {
        acked = tcp_slow_start(tp, acked);
        if (!acked)
            return;
    }
    /* In dangerous area, increase slowly. */
    tcp_cong_avoid_ai(tp, tp->snd_cwnd, acked);
}
```

那么然后呢？

有了这个拥塞控制算法，是不是就万事大吉了呢？从某种程度上说，是的！Why？？

我们知道，发包的终端是计算机，而早期计算机的处理器决定了发包的速率，这个时候，我们看一下网络骨干上的那些处理数据包转发的设备。

像思科这种厂商，他们的路由器，交换机的处理能力是发包的终端计算机难以企及的，专业的路由交换设备拥有更加高频的处理器，甚至有自己的线卡，这些高端设备完全可以Cover住任何计算机发送的数据包的转发，毫无压力。一切都在***存储转发***式的网络中有条不紊的进行着。

中间节点的高端设备只需要不多且固定数量的缓存，就可以暂时存储还没有来得及处理的数据包，需要多少缓存只需要通过排队理论的公式就能计算出来。

如果将路由器交换机看作是服务台，那么终端计算机发送的数据包按照泊松到达的原则到达并排队，然后被背靠背地服务并转发出去，这看起来非常美好！***如果接入网络的终端计算机更多更强了，只需要换更高端的转发设备即可***，这难道有什么问题吗？

我记得2004年我第一次接触H3C的讲座的时候，从最低端的几千元的设备一直到上百万的设备，都给我们看过并且摸过，当时特别惊讶，我在想，一台上百万的设备接在骨干网的通信机房，开机时，那将是多么壮美的场景，所以我一直都想去通信领域的企业工作，只可惜未能成行…

这场计算机网络终端和中间转发设备之间军备竞赛好像会促使他们中的任何一方持续进化成巨无霸，然而，它们忽略了一个事实！

那就是，***摩尔定律遇到了热密度的上限！***

这是一件非常令人遗憾的事情，面对这个无法突破的上限，中间转发设备被逼停在了山巅，眼睁睁看着计算机终端的处理器，网卡性能和自己的距离越来越近，却无法再前进一步！

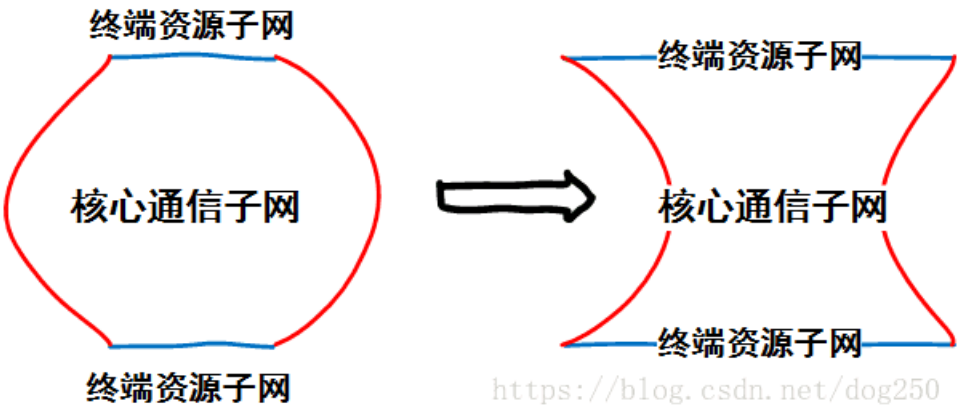
把时间拉到眼前，我们现在可以在普通的服务器甚至个人计算机上轻松安装最新的Intel万兆网卡，甚至40G的网卡，中间转发设备却再也无法甩开这些几个数量级了，毕竟有摩尔定律的大限在那！

《第二天》引文一段：

> 在演讲中，Van Jacobson 说互联网的这些已经发生了改变：在以前的互联网上，交换机可能总是拥有比服务器更快的网卡，所以这些位于互联网中间层的服务器也可能比客户端更快，并且并不能对客户端发送信息包的速率有多大影响。
> .
> 很显然今天已经不是这样的了！众所周知，今天的计算机相比于 5 年前的计算机在速度上并没有多大的提升（我们遇到了某些有关光速的问题）。所以我想路由器上的大型交换机并不会在速度上大幅领先于数据中心里服务器上的网卡。

抛开价格不谈，看看我们手里的手机，再看看2004年那些大型网络设备，看看它们之间的差距，是不是在缩小！

我们可以预见的是，随着计算机终端越来越追平中间转发设备的性能，即使是1988年提出的拥塞控制算法，对于真实的网络拥塞情况也将会越来越力不从心，网络变成了下面的样子：



1986年的拥塞崩溃还会重演，而且可能就在眼前！或者至少是不远的将来！

我在高中的时候，订阅过一本叫做《科学美国人》的杂志，大概是2001年一期，引出一个《互联网崩溃》的论题，当时正值第一次互联网爆发或者说泡沫时期，作者基于1988年Van Jacobson的论文里的拥塞控制算法，提出了一种担忧，作者担忧互联网将在10年内崩溃，即1988年的拥塞控制算法将在10年内失效！后来的事实表明，崩溃来得确实迟到了。

除了学术界，其实，从进入21世纪第一次互联网领域的寒武纪大爆发开始以来，上面的这个***互联网总有一天会拥塞崩溃***的问题也已经被网络设备厂商注意到了，虽然意识到自己面临了摩尔定律的极限，处理性能无法做进一步的提高，但是，横向的扩展还是可以继续的，***无法更快，但能更多***，无疑，***增加处理器和线卡的数量性价比远不如将来不及处理的数据暂时存起来***，这带来了一种解决方案：***增加缓存大小***！

排队理论和泊松到达原则预示着，缓存总是会被清空，这使得超大缓存队列的中间网络设备成了一种趋势。

这并没有解决问题，而是引入了问题，网络不再是一个***用完即走***的设施，而成了一个巨大的缓存设施，类似北京四环快速路那样，成了巨大的停车场！

对！***这就是Bufferbloat！！***

Bufferbloat将对数据包带来严重的延迟，降低用户体验！不过，好歹Bufferbloat也是有些可以借鉴的地方的，比如我觉得Kafka这玩意儿说白了就是一个Bufferbloat模型下的组件，你说Kafka这种设计好吗？反正我觉得挺不错的！这里给出一个Kafka的博客链接，作为备注：
《Kafka设计思想的脉络整理》：https://blog.csdn.net/dog250/article/details/79588437

那么，解决方案是什么？

《第二天》里说的很明确：***以更慢的速率发送更多的信息包以达到更好的性能***！

反正发快了也是排队，排队造成的延迟对于接收端而言和发得慢没有任何区别，不如用Pacing来平滑泊松到达的峰谷，还能减少甚至避免缓存爆满而造成的丢包。这就是Pacing慢发背后的逻辑。

核心骨干网络的高带宽等待用户计算机发送效率提升的时代已经结束了，目前大多数计算机发送数据包的速率已经超过了具有光速极限的传播速率，是时候要以带宽作为发送速率的基准了，以前是能发多快就发多快，核心交换设备应付低速的计算机终端不是个事儿，如今高速计算机终端则必须发送速率适应网络的处理能力，即***适应网络的瓶颈带宽***。

然后《第二天》又给出了一个具体的方案，那就是***“使用BBR”***！

> 在上面我说过：“假设你可以辨别出位于你的终端和服务器之间慢连接的速率……”，那么如何做到呢？来自 Google（Jacobson 工作的地方）的某些专家已经提出了一个算法来估计瓶颈的速率！它叫做 BBR，由于本次的分享已经很长了，所以这里不做具体介绍，但你可以参考 BBR：基于拥塞的拥塞控制 和 来自晨读论文的总结 这两处链接。

来自the morning paper的一篇blog值得推荐：
《BBR: Congestion-based congestion control》：https://blog.acolyer.org/2017/03/31/bbr-congestion-based-congestion-control/
当然了，你也可以看BBR的原始Paper或者，我写的一些博客，这里就不给出链接了。

我能说什么呢？如果我把2016年BBR发布的那一年作为TCP的第二个分水岭，你觉得合适吗？我觉得是合适的！

- **1974年-1988年**：第0代TCP，没有拥塞控制
- **1988年-2016年**：第1代TCP，引入Van Jacobson论文里AIMD模型的拥塞控制
  计算机终端不如核心交换设备，处理性能差距巨大，Burst发送。
- **2016年-至今/未来的某一年**：第2代TCP，引入Google的BBR拥塞控制并持续进化
  计算机终端处理性能追平核心交换设备，Burst发送会造成Bufferbloat，采用Pacing发送。

说来也巧了，上周日，正好总结了一幅图，展现拥塞控制算法的发展脉络：
《总结一幅TCP/QUIC拥塞控制(CC)算法的图示》：https://blog.csdn.net/dog250/article/details/81834855
《第二天》这篇文章是周一才推给我的，和我前一天写的这个基本在说同一个话题，这接力秒啊！

好了，现在我们从代码实现的角度看看代码是如何顺应历史的吧。

从第一行代码运行在第一台计算机上的时候，正如《第二天》所说：***网络代码被设计为运行得“尽可能快”！***，任何控制子系统都是这般，是否更快几乎是衡量一个算法优劣的首要标准！网络软件也不例外，如《第二天》文中所述：

> 所以，假设我们相信我们想以一个更慢的速率（例如以我们连接中的瓶颈速率）来传输数据。这很好，但网络软件并不是被设计为以一个可控速率来传输数据的！下面是我所理解的大多数网络软件怎么做的：
> .
> 1. 现在有一个队列的信息包来临；
> 2. 然后软件读取队列并尽可能快地发送信息包；
> 3. 就这样，没有了。

在计算机终端和网络核心交换设备处理性能差异巨大的年代，所有的计算机终端均采用***“尽可能快”***的方式发包时，这显然是***提高效率***的最佳做法。

对于计算机发送数据包而言，依照这个原则就是让数据包以最快的速度离开计算机，能多快就多快，事实上，当前的很多网卡(比如Intel x350…)的发送带宽已经超过了大多数的链路带宽。

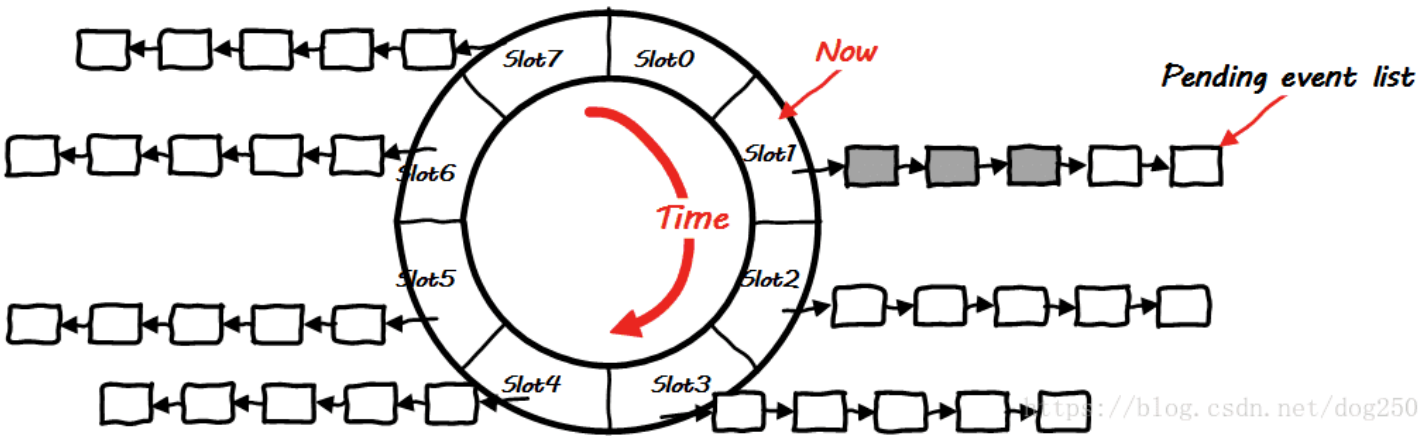
当核心网络交换设备遇到摩尔定律瓶颈，其性能提升不足以扛得住大量高速计算机终端快速发包带来的处理压力时，队列增长就是必然的，而我们知道，数据包排队现象是一种***毫无价值的临时避免丢包***的措施，而且平添了延时和存储成本，因而，降低计算机终端发包的速率变成了最有效最经济的措施。

《第二天》给出了解决方案：***一个更好的方式：给每个信息包一个“最早的出发时间”***!

好吧，具体到实现上，那就是所谓的***用时间轮盘替换队列！！！***。你不知道什么是时间轮盘？也许吧，但是你应该知道Linux内核网络数据包schedule模块FQ吧。是的，这个实现就是一个时间轮盘！

这个所谓的时间轮盘机制其实在Linux内核的TCP实现中已经有所使用了，比如针对TCP连接的超时重传Timer就是用时间轮盘实现的，不然的话，每一个包都启用一个Timer，开销甚大。说白了，时间轮盘就是把待定的Event进行排序，然后采用一个一维的时间序列顺序处理，思路非常简单！

抽象地讲，时间轮盘是下面的样子：



如此，数据包只需要在时间轮盘里找到一个位置，就可以实现Pacing发送了。

我贴出两篇之前写的博客的链接，里面有详细说FQ的一种时间轮盘实现的细节：
《Linux FQ 队列实现原理浅析》：**：https://blog.csdn.net/dog250/article/details/80025939
《合并N个有序链表与FQ公平调度》：**：https://blog.csdn.net/dog250/article/details/80234049

在传统的第0代，第1代TCP时代，所谓的“尽可能快地发送”，注定数据包的发送方式是突发的，即Burst方式，而当时间来到***我们已经意识到核心转发设备的极限从而对Burst发送方式有所收敛***的第2代TCP时代，我们必然要采用基于时间轮盘的Pacing发送方式以适应链路的瓶颈带宽！

在《第二天》这篇文章的最后，作者抒发了一种展望：

> 所以重点是假如你想大幅改善互联网上的拥塞状况，只需要改变 Linux 网络栈就会大所不同（或许 iOS 网络栈也是类似的）。这也就是为什么在本次的 Linux 网络会议上有这样的一个演讲！

或者说，这也许是我们大家的展望！

插叙一段。这里不得不说的是，Linux赢了。还记得当初有人比较Windows和Linux吗？然而过去了很多年后，请问你有多久没有开启家里的Windows机器了？是的，我们基本都是一部手机足矣，而我们的手机中有超过60%的，其底层，正是Linux内核。

微软无疑是PC时代的王者中的王者，曾经记得微软以Windows打Linux，微软以dot NET对抗Java，多线作战也未见颓势。然而在移动互联网时代，事情起了变化，你会发现Linux和Java似乎堆在一起了，这不就是Android吗？有点意思，微软从一打二变成了一打一，然而却更被动了，事实上，微软在一打一的时候，已经输了…

互联网依然在持续发展进化，作者在抒发了对这个变化的事实之一番感慨后，结束了《第二天》：

> 通常我以为 TCP/IP 仍然是上世纪 80 年代的东西，所以当从这些专家口中听说这些我们正在设计的网路协议仍然有许多严重的问题时，真的是非常有趣，并且听说现在有不同的方式来设计它们……等等，一直都在随着时间发生着改变，所以正因为这样，我们需要为 2018 年的互联网而不是为 1988 年的互联网设计我们不同的算法。

是啊，30年前TCP/IP的架构在现如今却依然可以拥抱变化，这也让我不禁感叹TCP/IP基因的优良，同时，还有一些同样拥抱变化的别忘了，那就是UNIX，以及我们人类文明自身！

关于《第二天》的读后感，我想就到此为止了，但正如本文一开始所说的，接下来还将有一个部分的内容，那就是关于Van Jacobson那篇经典论文的点点滴滴，如是解释一番后，方可不留遗憾地结束本文！

关于《Congestion Avoidance and Control》

再次给出链接：
《Congestion Avoidance and Control》：http://ee.lbl.gov/papers/congavoid.pdf

我说，没有读过这篇论文的，就别说自己是TCP拥塞控制圈里人了，这不是开玩笑。因为只有你读了这篇论文，才知道拥塞控制的基本原则是什么不是什么。

我再次重申，拥塞控制算法不是用来加速TCP的，相反，它是用来减速TCP的。想加速TCP，你必须修改控制状态机！

论文里提到

> The flow on a TCP connection (or ISO TP-4 or Xerox NS SPP connection) should obey a 'conservation of packets' principle. And, if this principle were obeyed, congestion collapse would become the exception rather than the rule. Thus congestion control involves finding places that violate conservation and fixing them.
> …
> **A new packet isn't put into the network until an old packet leaves**

嗯，就是这个原则，出去一个，进来一个。

在这个基础上，AIMD算法被提出来。AIMD其实也是见招拆招的产物，它本身就是为了Fix下面的违规：

> There are only three ways for packet conservation to fail:
> 1. The connection doesn't get to equilibrium, or
> 2. A sender injects a new packet before an old packet has exited, or
> 3. The equilibrium can't be reached because of resource limits along the path.

数据包守恒是拥塞控制的最最最最最最最最基本的原则，这个原则基于**单流单链路固定带宽**提出，非常直接且简单，很容易理解。

在现实实施中，我们知道，TCP是一个遵循端到端原则的协议，即它对网络是无感知的，所以它必须靠某种算法去动态适应**不固定的带宽**，比如任何时候都可能会有任何地方发出的新的TCP流，同样，任何时候都有可能有旧TCP流退出从而腾出新的带宽空间，如何去动态适应这些，是最终的算法必须要考虑的事情。

所以，最终，该论文提出了AIMD算法的三大构件：

- **1.总的原则**：单流固定带宽必须遵守数据包守恒，在此基础上，多流动态带宽的一般情形下—->
- **2.动态适应**：AI，即加性增窗探测
- **3.公平性**：MD，即乘性降窗收敛

其中，第2点和第3点在不断地博弈，**第2点显然为了fetch可能并没有的空余带宽而牺牲了公平性，而第3点则反过来为了公平性乘性降窗而牺牲了性能！**

注意，这其中有一个叫做Slow-Start的插曲，所谓的Slow-Start，其要义是***用比较快的速度上探网络的处理能力***，并不是真正的Slow。

一旦一个连接开始，拥塞控制的AIMD原则最终会让该连接收敛于与其它连接一致公平的位置，这样的算法就是***好的算法***，一个好的算法绝不是一个快的算法，在效率之前首先要考虑的是公平！

AIMD的过程实际上是一个合作过程，是一个所有的节点一起来公平填充网络交换节点缓存的过程，注意，AIMD的过程并没有涉及到***带宽***的概念，**整个过程就是对交换节点缓存的探测和考验**。因此，一旦缓存被填满，那么所有的途经于此的TCP连接均将**同时**面临丢包！以此为假设，我们来推导经典AIMD的公平性。

好的，接下来我来简单说一下AIMD为什么是收敛的，是公平的。

假设：
1. 缓存填满即拥塞，尾部丢包。
2. 拥有

AIMD探测周期		
第	…	…
第	…	…
…	…	…
第	…	…

从上面的这个过程中，很明显，最终

事实上，最终初始窗口

在算法收敛的过程中，我们可以看到初始窗口的影响是在***指数级衰减***的，这个过程横向铺开来看，就很像一个移动指数平均的过程。

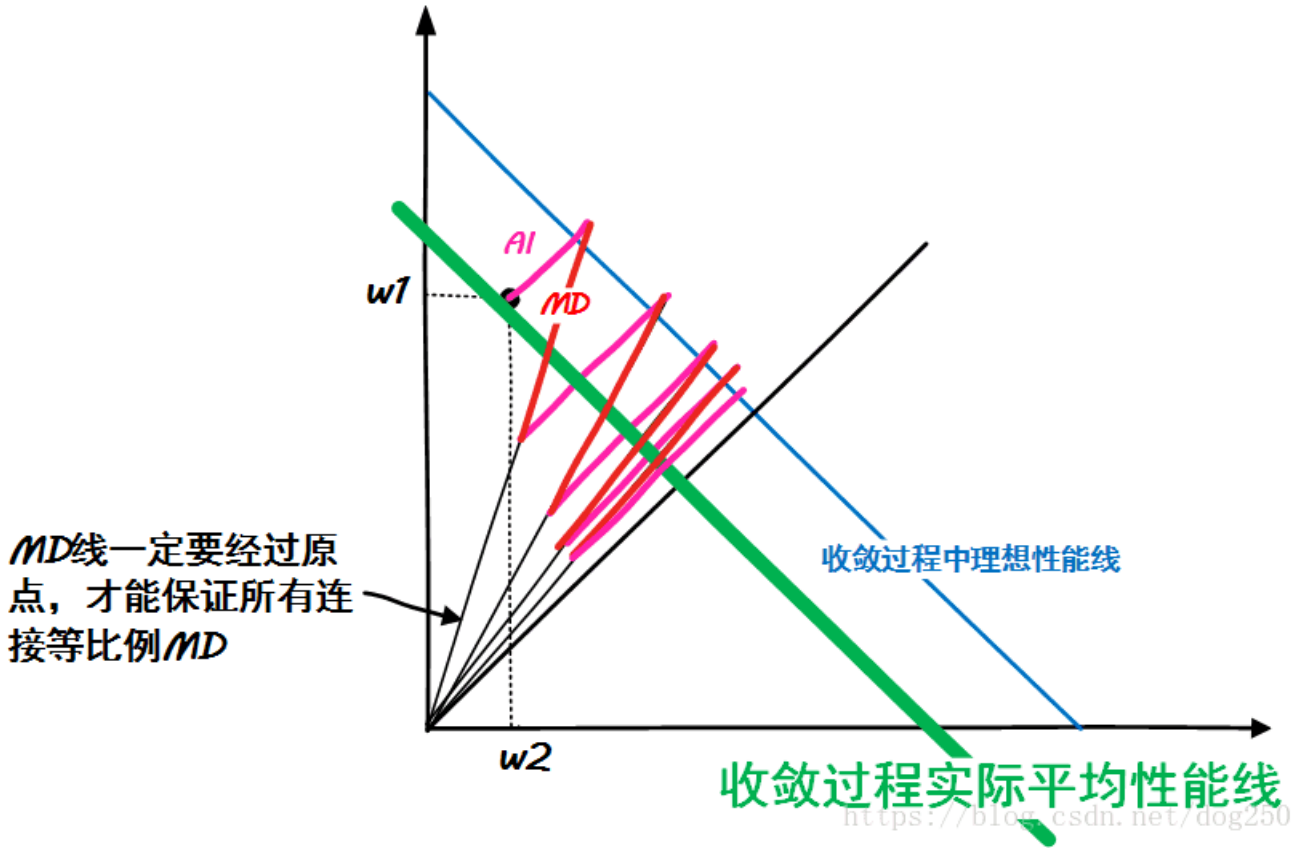
我记得在几年前的时候，我写过AIMD这方面的分析，比如下面这两篇：
《TCP核心概念-慢启动，ssthresh，拥塞避免，公平性的真实含义》：https://blog.csdn.net/dog250/article/details/51439747
《TCP自时钟/拥塞控制/带宽利用之脉络半景解析》：https://blog.csdn.net/dog250/article/details/51694591
其中，我主要描述了以下的主题：

- ssthresh的意义
- 拥塞的成因
- 公平性的推导

关于拥塞的成因，我用排队论理论分析过，其实就是泊松分布的到达率峰值偏置导致的，假设交换设备的处理能力为

而我们知道，所有的TCP都在进行着同样的AIMD过程，因此，堆积是不可避免的，也就是说，AIMD时代也就是***1988年后第1代TCP拥塞算法必然会导致拥塞***，Van Jacobson所谓的公平性也是建立在拥塞意义上的公平性。

下面还是这幅熟悉的图，即视觉上理解AIMD算法的最好的图：

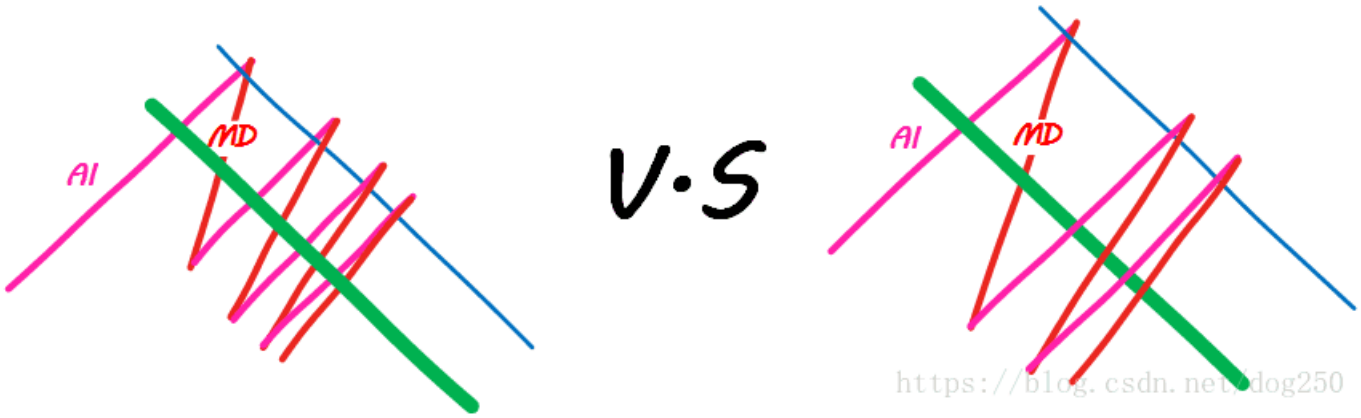


我们从这幅图里来看*效率和公平性之间的矛盾*！

首先，我们先来看如何来分别衡量效率和公平性：

- **效率**：绿色线和蓝色线之间的接近程度
- **公平性**：紫色，红色线的密集程度，即收敛速度

绿色线大致包络穿越了紫色，红色线段的中点的位置，如果绿色线越往上，说明收敛性能越接近理想，但是紫色，红色线段就会越密集，收敛到公平的时间就会越久，说明越不公平，反过来，紫色，红色线段越稀疏，则说明收敛到公平的周期越短，越公平，但是这样一样绿色线就会离开理想性能线越远！请看下图：



我感觉我说清楚了。*效率和公平是不能兼顾的*！完美！

为什么是AIMD？我前面用例子和图示分析了AIMD确实是OK的，但是，如果不是AIMD是不是不OK呢？

是的，AIMD是唯一的方案！

为什么？本文已经够长了，就不多说了，这里给出一点引子。这涉及到控制系统原理。因为***网络本身属于一种线性系统，而线性系统的平衡因子必然是指数的***，我刚刚说过，AI过程必然导致拥塞，也就是说拥塞是AI的结果，拥塞即不平衡状态，恢复到平衡状态则必然是加法的积累，即乘法。同样的原理可以解释为什么RTO需要指数级退避。

完结篇

也许你觉得我接下来要聊聊具体的算法实现细节了。不，我不会说的，因为已经说过了，从Reno到NewReno，再到BIC，再到CUBIC，最后我们看，BBR来也，这些我都说过了！

也许你以为BBR彻底颠覆了AIMD，但是非也！最终如果你跟踪BBR v2.0，你会发现BBR本质上也还是要靠向AIMD，否则将永远无法登堂入室。

我经常说，BBR之前的拥塞控制算法背后是一个***数学上的收敛模型***，而BBR没有这样的模型，BBR纯属工程产物，纯工程产物是无法预测的，只有数学模型可以预测和求证。那么其实，BBR背后的模型为什么不能同样使用AIMD呢？

本文对《第二天》发表了一些观点，又分析了《Congestion Avoidance and Control》中AIMD的细节，当然有很多细节还是没有涉及到，比如RTT，RTO的计算公式为什么长那个样子，其实Van Jacobson也是有提到的，因为RTT和RTO计算的正确性将直接影响数据包守恒判断的准确性！这一点我没有分析，感兴趣还是去读原论文吧。

嗯，本文终于就这么结束了，下面一个小节，例行的，一些感慨，包括我前几天微信朋友圈发的一些文字…

写在最后

引用一段我前天发的朋友圈：

> 在一个网络专家群跟人聊天，get 到几个points，分享出来。
> 有人问我为什么喜欢研究历史，特别是欧洲历史。我的答案其实很简单，我只是想形成一种认知方法论而不是仅仅为了得到一些所谓的知识，更不是为了显得自己很有文化。
> 当然，我并非什么特别牛的大神级人物，我只是比较喜欢探讨并分享一些平时大家忽略的东西罢了。我的观点很明确，一位大师和一位专家之间区别是，大师get到的是一种认知，而专家get到的则仅仅是一种技能，各个行业各个领域都是如此。所以大师可以在对待任何问题的时候举一反三游刃有余，而专家一般在自己的领域外则歇菜。另外一层意思，大师会把自己的认知延续到生命的每个维度，而专家则只是将技能作为谋生的手段。
> 我认识很多专家，有自诩的，也有大家封神的，有天生聪明绝顶的，也有熬年头熬出来的。以TCP协议为例，能说出细节的铺天盖地，然而能简述为什么这么设计的却寥寥无几，当我问问题的时候，我当然是希望讨论者可以从分组交换网伊始的1969年说起，从ARPA说起，每一步的见招拆招成全了汗牛充栋的Changelog，然后，就是我们现在使用的TCP。。。如果让我解释弹道导弹，我一定会感谢第一个学会用树枝弹石子打鹿的原始人。。。所谓日光之下并无新事，便是我的认知。回到TCP，如果你不知道1986年的网络崩溃是如何发生的，你就不可能理解1988年雅各布森的论文，进而无论你多么熟悉CUBIC算法的实现你也无法理解这一切背后的逻辑，最终可能在这方面所有的工作都是胡扯，也就无法真正理解BBR，更别提去设计BBR v3了。我是愿意花大量时间去梳理认知的，如果你不理解罗马帝国和诸日耳曼民族的在公元元年到五世纪的对峙，你就不能理解1517年以后的宗教对立，你会错过维斯特伐利亚条约的意义，你就不能真正理解英国和美国的真正角色，因为你不知道大移民的动因。。。如今任何人靠看或真或假的新闻就能知道特朗普的政策，但是很少有人能解释这一切，就更别提预测，因为大部分人都没有形成一种连续的认知，你能用术语描述细节，但这并不意味着你能解释why。。。
> 还好，有幸能有人跟我讨论这一切，非常感谢，我也会分享我的认知。

具体我就不评论了，大致就是这个意思，自己评论自己的东西也挺无趣的。本文也是因为这个朋友圈而写的(当然，这个朋友圈文字并不是全部因素)，做个备忘，不多说。

最后是比较感谢小小语言班的两位美女老师的，带小小去欧洲这么久帮忙照顾，昨天晚上特意请她们吃了深圳还不错的一家日料(甩大渔几条街的那种，当然和上海静安长宁的店是没法比，深圳还算不错)，美女照片就不上了，上个大盘吧(角度不好，好位置在对面，注意对面的三种大虾)：

注解：这家店我去过好几次了，认准***【食冶日本料理铁板烧(南山海岸城店)】***，之前都是朋友请的，这次是请别人...十分推荐这家店，环境好，食材好，比温州老板的店好太多。

... 浙江温州，皮鞋湿！！