TDP（TCP-over-UDP library）：基于 UDP 协议之上实现通用、可靠、高效的 TCP 协议

黄洪波(huanghongbo@gmail.com)
http://blog.csdn.net/huanghongbo/

一、介绍

随着互联网应用广泛推广，出现了越来越多的网络应用，其中基于p2p 思想的各种网络技术的产品也越来越多的出现在我们的视野当中。从最早闻名的Napster 到现在的 Bittorrent、eMule、skype 等产品，P2P 这种网络应用模式已经从各个方面深入人心。这些产品在各自的网络实现技术上，都以各自的方法解决着同样面临的一个问题，如何让他们的软件产品在各异的网络拓扑结构中顺利的进行 P2P 通信。

众所周知，在当今的网络拓扑结构中，普遍存在使用 NAT 设备来进行网络地址转换，而让应用程序能跨越这些 NAT 设备进行全双工的通信，就成为非常重要的一个问题。对于实现跨越 NAT 通信可以采取很多种办法（对于能够直接连接、反向连接的情况不在此列）：首先是通过服务器进行转发，这是比较粗暴的方法，而且在用户量大的时候，转发服务器需要付出相当大的代价；第二，可以使用 NAT 穿透技术。而大家知道关于 NAT 穿透中，UDP 穿透的成功率比起 TCP 穿透要高出许多，这一点这里将不做多述，可以参考 Bryan Ford 的文章《Peer-to-Peer Communication Across Network Address Translators》（http://www.brynosaurus.com/pub/net/p2pnat/）。因此在 UDP 协议上构建一些大型的网络应用程序可能会成为很多人的需求。

当然也可能基于更多的原因，会有很多人希望能在UDP 协议上进行大型应用程序的构建。然而 UDP 协议本身存在着不通信不可靠的缺点，于是对于基于UDP 进行可靠通信的需求就浮现出来了。目前在网络上有许多人正做着这一工作，UDT、RakNet、eNet 等都是构建在 UDP 之后网络可靠通信开发库。然后这些库开发时都针对了一些特殊应用来进行设计的，不具备通用性。比如 RakNet 是为游戏应用而设计，对于实时性等游戏相关的网络需求有很好的支持，对于大批量数据传输却有点力所不及。而 UDT 基于一种基于带宽速率控制的拥塞控制算法进行设计（http://udt.sourceforge.net/doc/draft-gg-udt-01.txt），主要用在小数量的 bulk 源共享富裕带宽的情况下，最典型的例子就是建立在光纤广域网上的网格计算，而在 ISP 提供带宽有限的情况下运行却显得消耗资源并性能不足。甚至可能被防火墙，或ISP 服务商判断为恶意带宽使用攻击。这些都使用得他们不能被广泛地用于各种网络应用程序。另外大家也陆续发现目前的 UDT 实现版本存在的一些问题。比如 UDT 做服务端接收连接时，总是新开一个端口与客户端进行连接，这样会带来几个问题：1）较多客户端连接上来时，服务端新打开的众多端口中可能有的端口会被防火墙拦截而导致通信失败，2）如果客户端处于 Symmetric NAT 和 Port-Restricted Cone NAT 后面时，将导致服务器端与客户端连接无法成功建立，3）由于 udp 端口数最大值有限，所以 UDT 服务器端可接收的连接数也因些受限。再有就是不仅仅是 UDT 库，基本上所有的 UDP-based 可靠通信库，都未提供穿越 proxy 代理的功能（socks5）；再有就是对 UDP 打洞技术有的支持得不完善或并不支持。

基于这些原因，使得我需要开发一个基于 UDP 协议之上实现一个可靠、高效、通用的通信库，来满足我目前所开发的项目的需要。TCP 协议算法已经是经过多方面及多年的验证，是最具通用性，且可靠高效的。虽然 UDT 等各种库指出 TCP 在这样或那样的网络环境下存在

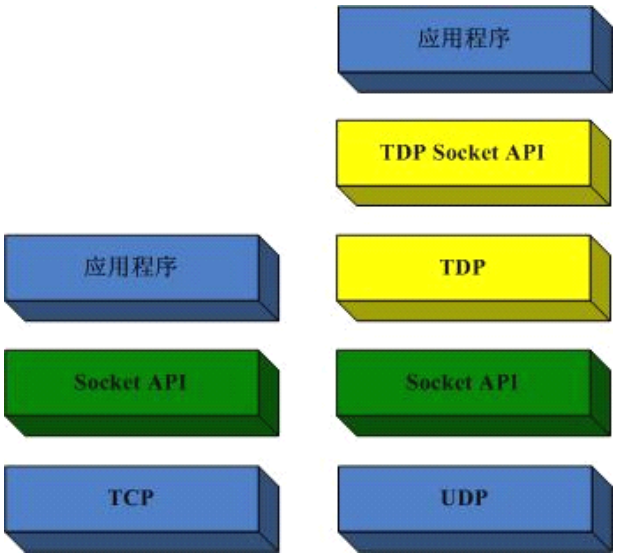
不足，但众多实现当中他仍然是最通用、可靠、高效的。相信有许多人跟我一样，需要这么一个开发库，所以我打算在开发过程中，陆续公开相关的文档及这个开发库。

二、设计目标

TDP 主要的目标就是在 UDP 层之上实现 TCP 的协议算法，使得应用程序能够在 UDP 层之上获得通用、可靠、高效的通信能力。

TDP 网络开发库所实现的算法，都来自久经考验的 TCP 协议算法，网上有着非常多的参考资料。在实现当中，参考最多的是 Richard Stevens 的《TCP/IP 详解》。

TDP 提供的用于开发的应用程序接口与 Socket API 非常相像，姑且称之为 TDP Socket API，基本上的函数名与参数等都与 Socket API 相一致，但是 TDP Socket API 的 API 接口都位于命名空间 TDP 当中。只要使用过 Socket API 进行开发过的朋友，将都会使用 TDP 库进行开发。下图为 TDP 及 TDP Socket API 所处在的协议栈应用中的位置，以及与 TCP 协议栈应用的对比。



三、协议说明

1. 协议格式

TDP 的实现的算法虽然与 TCP 实现的算法是大致相同的,但 TDP 的协议格式只是从 TCP 协议格式获得参考，但并不完全与他相同。TDP 的协议格式如下：

接下来介绍一下协议格式的各个字段含义。

4 位首部长度：表示用户数据在数据包中的起始位置。

LIV：连接保活标志，用于表示 TDP 连接通路存活状态。

ACK：确认序号有效。

PSH：接收方应该尽快将这个报文段交给应用层。

RST：重建连接。

SYN：同步序号用来发起一个连接。

FIN：发端完成发送任务。

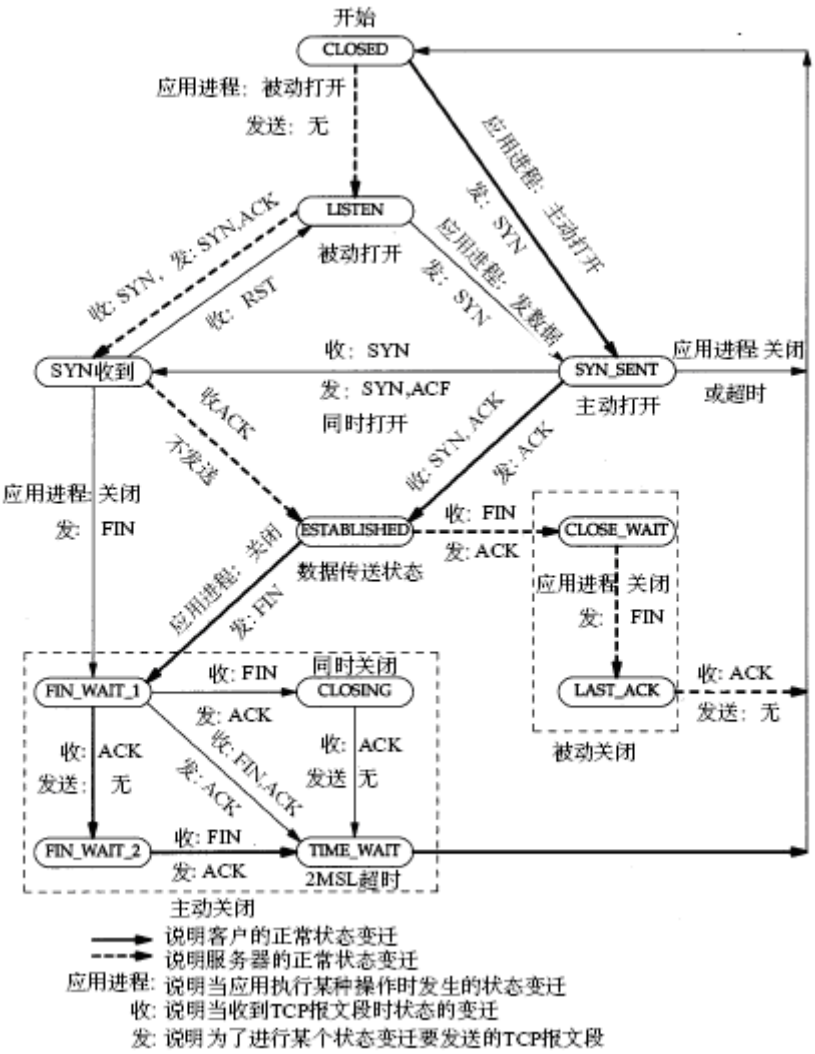
16 位窗口大小：接收端可接收数据的窗口大小。

选项：只有一个选项字段，为最长报文大小，即 MSS。TDP 选项格式与 TCP 选项格式一致，kind=0 时表示选项结束，kind=1 时表示无操作，kind=2 时表示最大报文段长度。如下图：

数据：用户通过 TDP 传输的数据。

2．TDP 连接建立与终止

TDP 的连接建立与终止可以参考 TCP 的状态变迁图(此图的详细解释请参考《TCP/IP 详解 卷一》第 18 章)，如下：



2.1 连接建立

2.1.1 三次握手

连接建立分要经过三次握手过程：1)客户端发送一个 SYN 段到指明客户打算连接的服务器的端口，报文段中要设置客户端初始序号。2）服务器发回包含服务器的初始序号的 SYN 报文段作为应答。同时，将确认序号设置为客户的初始序号加 1,并设置 ACK 位标志报文段为确认报文段。3）客户端必须将确认序号设置为服务器初始序号加 1,对服务器的 SYN 报文段进行确认。

TDP 在全局维护一个初始序号种子，这个初始序号为随时产生的 32 位整数。

连接建立的超时和重传初始值为 3 秒，超时采用指数退避算法，3 秒超时后超时值为 6 秒，然后是 12 秒，24 秒……。连接建立最长时间限制为 75 秒。

2.1.2 NAT UDP PUNCH 模式

当 TDP 工作模式是 NAT UDP PUNCH 时，在三次握手之前，向对端 NAT 端口及预测端口间隔默认 2ms 发送默认为 10 个 LIV 报文段，一来用于打开自己的 NAT 端口，二来是用于进入对端 NAT 端口。默认值可以由用户程序设置。这时的 LIV 报文段中初始序号及确认序号都为 0。

当接收到对端LIV报文段后，立即停止LIV报文段发送，发出SYN报文段进行连接建立。这时有两种可能：其一是另一端直到接收到该SYN报文段之前，都没有接收到LIV报文段，或是刚接收到但没有来得及发送SYN报文段，此时将会如上文描述的正常模式下连接建立的过程一致，将经历三次握手。基二是另一端在接收到该SYN报文段之前，也已经发送出SYN报文段，此时双方都需要对SYN报文段进行确认，可以称之为四次握手。

2.1.3 最大传输报文大小（MSS）

TCP报文段在连接建立时需要通报MSS，在TDP的实现中也进行通报，默认通报为1460字节（符合以太网标准，这个默认值允许20字节的IP首部、8字节的UDP首部和12字节的TDP首部，以适合 1500字节的IP数据报）默认值可以由用户程序设置。

TCP在对端地址为非本地IP时，默认通报为536字节。TDP之所以默认通报为1460，是因为TDP在数据传输过程中，实现了路径MTU发现技术，通过实际发现的MTU，进行MSS的动态调整，以尽量避免报文段在网络中的传输产生分片的情况。路径MTU发现技术在传输数据流一节中进行描述。

2.1.4 半打开连接及连接保活

半打开连接是指对端异常关闭，如网线拔掉、突然断电等情况将引发一端导演关闭，而另一端的连接却仍然认为连接处于打开当中，这种情况称之为半打开连接。TDP中的一个TDP SOCKET描述符由本地IP、本地端口、远端IP、远端端口唯一确定。当远端客户端连接请求到来时，服务端将接收到一个新的TDP SOCKET描述符，当这一个描述符唯一确定信息已经存在时，对新的连接请求发送RST报文段，通知其重置连接请求。对于旧的连接，由保活机制自动发现是否为半打开连接，如果是半打开连接，则自动关闭该连接。这里RST报文段与TCP中的RST报文段有些不一样，TCP的RST报文段工作描述请参考《TCP/IP详解 卷一》。

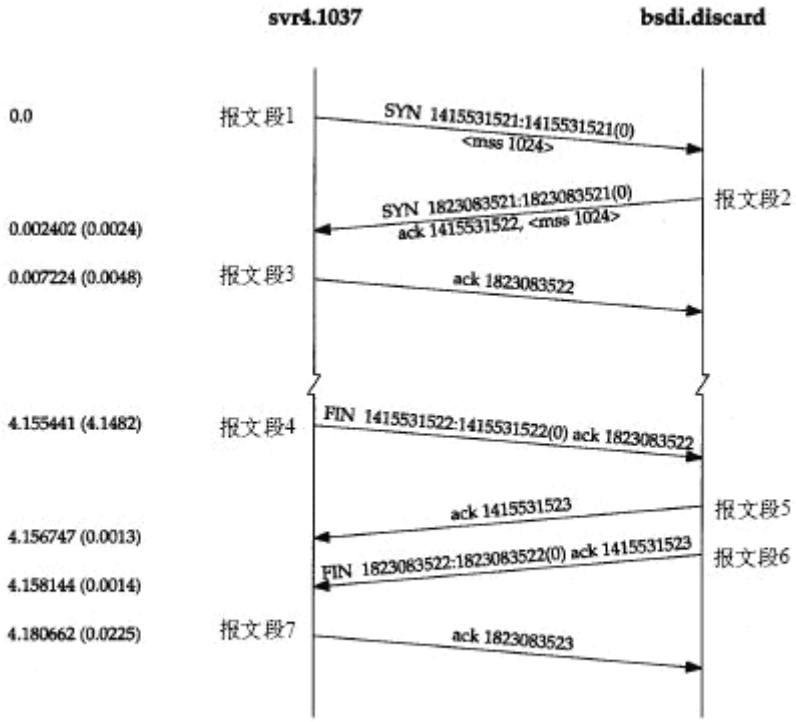
连接建立之后，TDP连接需要启动保活机制。TCP连接在没有数据通信的情况下也能保持连接，但TDP连接不行。TDP连接在一定时间段内如果没有数据交互的话，将主动发送保活LIV报文段。这个时间段根据TDP连接工作模块不同有所差异，在NAT UDP PUNCH模式下，这个时间段默认值为1分钟（大多数的NAT中，UDP会话超时时间为2－5分钟左右）；而在常规模块下这个时间段默认值为5分钟。默认值可以由用户程序设置，用户程序需要指明两种模块下的保活时间周期。这里TDP的保活机制与TCP中的保活机制完全不一样，TCP的保活机制描述请参考《TCP/IP详解 卷一》。

2.2 连接关闭

TDP连接与TCP连接一样是全双工的，因此每个方向必须单独地进行关闭。客户机给服务器一个FIN报文段，然后服务器返回给客户端一个确认ACK报文，并且发送一个FIN报文段，当客户机回复ACK报文后（四次握手），连接就结束了。

TDP连接的一端接收到FIN报文段时，如果还有数据要发送，需要继续将数据进行发送完成，然后才发出FIN报文段；如果还有数据未从缓存中取出，将取出数据，并进行确认，直到所有确认完成之后，然后才发出FIN报文段（此时如果有乱序的报文段情况不进行处理）。上面的描述也表现出，TDP是支持半关闭的，当一端发出FIN报文段时，仍然允许接收另一端数据。但是半关闭可能导致连接永远停留在状态图中FIN_WAIT_2状态中，此时保活机制仍然在工作当中，如果对端已经关闭，那么保活机制将在检测到时立即关闭这一连接。

下图是一个典型的连接建立与连接关闭的示意图，此图摘自《TCP/IP详解 卷一》。



连接建立与终止的时间系列

四、TDP 传输数据流

1．传输的报文段

在 TDP 工作过程中传输的所有报文段，只有 SYN 报文段、FIN 报文段、数据报文段是可靠的之外，其它报文段如 ACK 报文段、LIV 报文段、RST 报文段等都不是可靠的。SYN 报文段与 FIN 报文段传输中都占用一个序号，数据报文段在传输中根据传输的数据字节数占用相应的序号，其它报文段不占用传输序号。

成功接收数据报文段，应当将按序对下一个期望的数据报文段的序号作为确认序号发送 ACK 报文段进行确认。当出现接收到乱序的数据报文段时，将乱序数据报文段按序缓存，并发送期望报文段的 ACK 报文段进行确认。ACK 报文段的发送并非即时的，也并非是对应接收数据报进行一对一确认发送。ACK 报文段由 200ms 定时触发发送，也就是说 ACK 报文段要经受最多 200ms 的时延进行发送。ACK 报文段对此时期望的数据序号进行确认，因此并不是与接收数据报相对应。ACK 报文段是不可靠的，当丢失时对端将无法了解接收情况，因此发送方将会有一个超时机制，如果发现确认的 ACK 报文段超时，发送方将重发该数据报，这一点在第五节进行详细描述。

2．路径 MTU 发现及 MSS 通告

前面已经提到要在连接建立过程中会通告初始MSS，这个值可以由用户程序进行设置。但这个初始值是一个静态的。当通信的两个端点之间跨越多个网络时，使用设置的MSS 进行报文段发送时，可能导致传输的 IP 报文分片情况的产生。为了避免分片情况的产生，TDP

在数据传输过程中进行动态的路径MTU发现，并进行MSS的更新及通告。

TDP创建UDP SOCKET时，即将描述符设置IP选项为不允许进行分片（setsockopt (clientSock, IPPROTO_IP, IP_DONTFRAGMENT,(char*)&dwFlags, sizeof(dwFlags))）。在发送数据时以当前MSS大小值进行数据发送，如果返回值为错误码WSAEMSGSIZE（10040）表示为报文段尽寸大于MTU，需要进行IP分片传输。此时，缩减MSS大小再次进行报文段发送，直至不再返回错误码WSAEMSGSIZE（10040）。当MSS变更并能成功发送报文段后，需要向对端通报新的MSS值。每次MSS缩小后，默认隔30秒，TDP将默认扩大MSS大小，以检查是否路径MTU增大了（默认值可以由用户程序设置），之后隔30*2秒、30*2*2秒进行检测，如果三次都未发现MTU增大则停止进行检测。见RFC1191描述，网络中MTU值的个数是有限的，如下图描述（摘自RFC1191）。因此MSS的扩大及缩减，可依据一些由近似值按序构成的表，依照此表索引进行MSS值的扩大与缩减计算。

Plateau	MTU	Comments	Reference
	65535	Official maximum MTU	RFC 791
	65535	Hyperchannel	RFC 1044
65535			
32000		Just in case	
	17914	16Mb IBM Token Ring	ref. [6]
17914			
	8166	IEEE 802.4	RFC 1042
8166			
	4464	IEEE 802.5 (4Mb max)	RFC 1042
	4352	FDDI (Revised)	RFC 1188
4352 (1%)			
	2048	Wideband Network	RFC 907
	2002	IEEE 802.5 (4Mb recommended)	RFC 1042
2002 (2%)			
	1536	Exp. Ethernet Nets	RFC 895
	1500	Ethernet Networks	RFC 894
	1500	Point-to-Point (default)	RFC 1134
	1492	IEEE 802.3	RFC 1042
1492 (3%)			
	1006	SLIP	RFC 1055
	1006	ARPANET	BBN 1822
1006			
	576	X.25 Networks	RFC 877
	544	DEC IP Portal	ref. [10]
	512	NETBIOS	RFC 1088
	508	IEEE 802/Source-Rt Bridge	RFC 1042
	508	ARCNET	RFC 1051
508 (13%)			
	296	Point-to-Point (low delay)	RFC 1144
296			
68		Official minimum MTU	RFC 791

Table 7-1: Common MTUs in the Internet

TDP中MSS与MTU之间关系的计算公式如下：

MSS = MTU - 20(IP首部) - 8(UDP首部) - 12(TDP首部)。

3. Nagle算法

有些人误认为经受时延的捎带ACK发送是Nagle算法，其实不是。经受时延的捎带ACK发送是TCP的通常实现，在TDP中也是如此。而Nagle算法是要求一个TCP（TDP也是如此）

连接上最多只能有一个未被确认的未完成的报文段，在该报文段的确认到达之前不能发送其他的报文段。相反，TCP（TDP 也是如此）在这个时候收集这些报文段，关在确认到来时合并作为一个报文段发送出去。Nagle 算法对于处理应用程序产生大量小报文段的情况，有利于避免网络中由于发送太多的包而过载（这便是发送端的糊涂窗口综合症，关于糊涂窗口综合症在下文将做更详细描述）。

Nagle 算法适用于产生大量小报文段的情况，但有时我们需要关闭 Nagle 算法。一个典型的例子是 X 窗口系统服务器：小消息（鼠标移动）必须无时延地发送，以便为进行某种操作的交互用户提供实时的反馈。

默认的 TDP 实现中 Nagle 算法是关闭的，用户程序可以设置打开它。

4．窗口大小通告与滑动窗口

双方接收模块需要依据各自的缓冲区大小，相互通告还能接受对方数据的尺寸。双方发送模块则必须根据对方通告的接收窗口大小，进行数据发送。这种机制称之谓滑动窗口，它是 TDP 接收方的流量控制方法。它允许发送方在停止并等待确认前可以连续发送多个分组（依据滑动窗口的大小），由于发送方不必每发一个分组就停下来等待确认，因此可以加速数据的传输。

参照《TCP/IP 详解 卷一 20.3 滑动窗口》一节，滑动窗口在排序数据流上不时的向右移动，窗口两个边沿的相对运动增加或减少了窗口的大小，关于窗口边沿的运动有三个术语：窗口合拢（当左边沿向右边沿靠近）、窗口张开（当右边沿向右移动）、窗口收缩（当右边沿向左移动）。RFC 文档强烈建议不要在实现当中出现窗口收缩的情况出现，在我们的实现中也将不会出现。

当遇到快的发送方与慢的接收方的情况时，接收方的窗口会很快被发送方的数据填满，此时接收方将通告窗口大小为 0，发送方则停止发送数据。直到接收方用户程序取走数据后更新窗口大小，发送方可以继续发送数据；另外，因为 ACK 报文段有可能丢失，发送方可能没有成功接收到更新的窗口大小，因此发送方将启动一个坚持定时器，当坚持定时器超时，发送方将发送一个字节的数据到接收方，尝试检查窗口大小的更新。

在 Nagle 算法中接到过糊涂窗口综合症，在这里要进一步进行描述。糊涂窗口综合症是指众多少量数据的报文段将通过连接进行交换，而不是满长度的报文段，这将导致连接占用过多带宽，降低传输速率。糊涂窗口综合症产生是分两端的，接收方可以通告一个小的窗口（而不是一直等到有大的窗口时才通告），发送方也可以发送少量的数据（而不是等待其他的数据以便发送一个大的报文段）。要以采用如下方法避免这一现象：

1）接收方不通告小窗口。通常的算法是接收方不通告一个比当前窗口大的窗口（可以为 0），除非窗口可以增加一个报文段大小(也就是将要接收的 MSS)或者可以增加缓存空间的一半，不论实际有多少。

2）发送方避免出现糊涂窗口综合症的措施是只有以下条件之一满足时才发送数据：(a)可以发送一个满长度的报文段；(b)可以发送至少是接收方通告窗口大小一半的报文段；(c)可以发送任何数据并且不希望接收 ACK（也就是说，我们没有还未被确认的数据）或者该连接上不能使用 Nagle 算法。

5．PUSH 标志

PSUH 标志的作用是发送方使用 PUSH 标志通知接收方将所收到的数据全部提交给接收进程。在 TDP 实现中，用户程序并不需要关心 PUSH 标志。因为 TDP 实现从不将接收到的数据

推迟交付给用户程序，因此这个标志在TDP的实现中是被忽略的。

五、TDP超时与重传

1．带宽时延乘积与拥塞

每个网络通道都有一定的容量，可以计算通道的容量大小：

Capacity(bit) = bandwidth(b/s) * round-trip time(s)

这个值一般称之为带宽时延乘积。这个值依赖于网络速度和两端的RTT，可以有很大的变动。不论是带宽还是时延均会影响发送方与接收方之间通路的容量。

当数据到达一个大的网络通道并向一个小的网络通道发送，将发生拥塞现象。另外当多个输入流到达一个路由器，而路由器的输出流小于这些输入流的总和时也会发生拥塞。TDP超时与重传机制刚采用TCP的拥塞控制算法来进行发送端的流量控制。

2．往返时间与重传超时时间测量

超时与重传中最重要的部分就是对一个给定连接的往返时间（RTT）的测量。由于路由器和网络流量均会发生变化，因此一般认为RTT可能经常会发生变化，TDP应该跟踪这些变化并相应地改变相应的超时时间。

首先是必须测量在发送一个带有特别序号的字节和接收到包含字节的确认之间的RTT。由于数据报文段与ACK之间通常没有一一对应的关系，如下图（摘自《TCP/IP详解 卷一》图20.1）中，这意味着发送方可以测量到的一个RTT，是在发送报文段4和接收报文段7之间的时间，用M表示所测量到的RTT。

根据[Jacobson 1988]描述（见《TCP/IP详解 卷一》参考文献），用A表示被平滑的RTT（均值估计器），用D表示被平滑的均值偏差，用Err表示刚得到的测量结果M与当前RTT估计器之差，则可以计算下一个超时重传时间（用RTO表示下一个超时重传时间）。

A = 0 （未进行测量往返时间之前，A的初始值）

D = 3 （未进行测量往返时间之前，D的初始值）

RTO = A + 2D = 6 （未进行测量往返时间之前，RTO的初始值）

A = M + 0.5（第一次测量到往返时间结果，对RTT估计器计算初始值）

D = A / 2 （第一次测量到往返时间结果，对均值偏差D计算初始值）

RTO = A + 4D （第一次测量到往返时间结果，对均值偏差RTO计算初始值）

之后的计算方法如下：

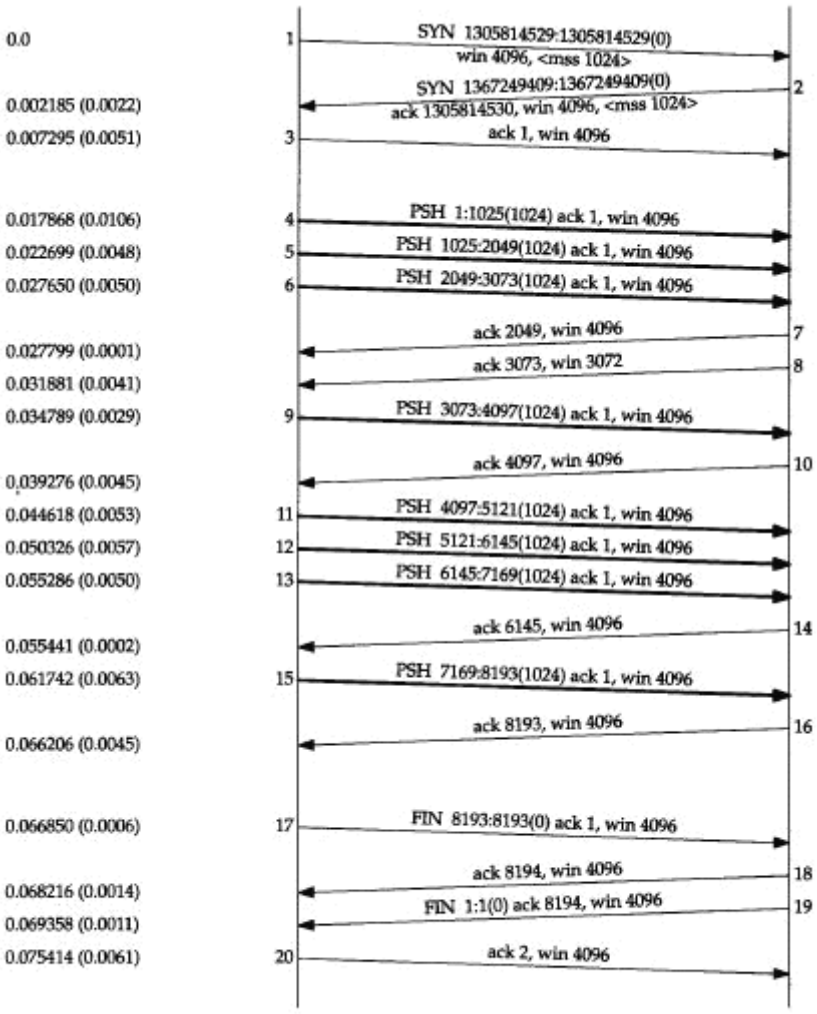
Err = M - A

A <- A + gErr

D <- D + h(|Err| - D)

RTO = A + 4D

其中g是常量增量，取值为1/8(0.125)；h也是常量增量，取值为1/4(0.25)。



从svr4 传输8192个字节到bsdi

Karn 算法：Karn 算法是解决所谓的重传多义性问题的。[Karn and Partridge 1987]规定（见《TCP/IP 详解 卷一》参考文献），当一个超时和重传发生时，在重传数据的确认最后到达之前，不能更新 RTT 估计器，因为我们并不知道 ACK 对应哪次传输（也许第一次传输被延迟而并没有被丢弃，也有可能是第一次传输的ACK 被延迟丢弃）。并且，由于数据被重传，RTO 已经得到了一个指数退避，我们在下一次传输时使用这个退避后的RTO。对一个没有被重传的报文段而言，除非收到了一个确认，否则不要计算新的RTO。

在任何时候对每个连接并行仅测量一次RTT 值，在发送一个报文段时，如果给定连接的定时器已经被使用，则该报文段不被计时，反之如果给定连接的定时器未被使用，则开始计时以测量 RTT 值。即并非每个发出报文段都进行测量RTT 值，同一时间段里只能有一个 RTT 值测量行为进行，不会并行进行多个 RTT 值测量。

3．慢启动

如果发送方一开始便向网络发送多个报文段，直至达到接收方通告窗口大小为止。当发送方与接收方在同一局域网时，这种方式是可以的。但如果在发送方与接收方之间存在多个路由器和速率较慢的链路时，就可能出现问题。一些中间路由器必须缓存分组，并有可能耗尽存储器的空间，将来得降低 TCP 连接的吞吐量。于是需要一种叫“慢启动”的拥塞控制算

法。

慢启动为发送方增加一个拥塞窗口，记为cwnd，当与另一个网络的主机建立连接时，拥塞窗口被初始化为1个报文段。每收到一个ACK，拥塞窗口就增加一个报文段（cwnd以字节为单位，但慢启动以报文段大小为单位进行增加）。发送方取拥塞窗口与通告窗口中的最小值作为发送上限。拥塞窗口是发送方使用的流量控制，而通告窗口是接收方使用的流量控制。

发送方开始时发送一个报文段，然后等待ACK。当收到该ACK时，拥塞窗口从1增加到2，即可以发送两个报文段。当收到这两个报文段的ACK时，拥塞窗口就增加为4。这是一种指数增加的关系。

4. 拥塞避免

慢启动算法增加拥塞窗口大小到某些点上可能达到了互联网的容量，于是中间路由器开始丢弃分组。这就通知发送方它的拥塞窗口开得太大。拥塞避免算法是一种处理丢失分组的方法。该算法假定由于分组受到损坏引起的丢失是非常少的（远小于1%），因此分组丢失就意味着在源主机和目标主机之间的某处网络上发生了拥塞。有两种分组丢失的指示：发生超时和接收到重复的确认。拥塞避免算法与慢启动算法是两个独立的算法，但实际中这两个算法通常在一起实现。

拥塞避免算法和慢启动算法需要对每个连接维持两个变量：一个拥塞窗口cwnd和一个慢启动门限ssthresh。算法的工作过程如下：

1) 对一个给定的连接，初始化cwnd为1个报文段，ssthresh为65535个字节。

2) TCP输出例程的输出不能超过cwnd和接收方通告窗口的大小。拥塞避免是发送方使用的流量控制，而通告窗口则是接收方进行的流量控制。前者是发送方感受到的网络拥塞的估计，而后者则与接收方在该连接上的可用缓存大小有关。

3) 当拥塞发生时(超时或收到重复确认)，ssthresh被设置为当前窗口大小的一半(cwnd和接收方通告窗口大小的最小值，但最少为2个报文段）。此外，如果是超时引起了拥塞，则cwnd被设置为1个报文段（这就是慢启动）。

4) 当新的数据被对方确认时，就增加cwnd，但增加的方法依赖于我们是否正在进行慢启动或拥塞避免。如果cwnd小于或等于ssthresh，则正在进行慢启动，否则正在进行拥塞避免。慢启动一直持续到我们回到当拥塞发生时所处位置的半时候才停止（因为我们记录了在步骤2中给我们制造麻烦的窗口大小的一半），然后转为执行拥塞避免。

慢启动算法初始设置cwnd为1个报文段，此后每收到一个确认就加1。这会使窗口按指数方式增长：发送1个报文段，然后是2个，接着是4个……。拥塞避免算法要求每次收到一个确认时将cwnd增加1/cwnd。与慢启动的指数增加比起来，这是一种加性增长。我们希望在一个往返时间内最多为cwnd增加1个报文段（不管在这个RT T中收到了多少个ACK），然而慢启动将根据这个往返时间中所收到的确认的个数增加cwnd。

处于拥塞避免状态时，拥塞窗口的计算公式如下（引公式参照BSD的实现，segsize/8的值是一个匹配补充量，不在算法描述当中）：

```
cwnd <- cwnd + segsize * segsize / cwnd + segsize / 8
```

5. 快速重传与快速恢复

由于我们不知道一个重复的ACK是由一个丢失的报文段引起的，还是由于仅仅出现了几个报文段的重新排序，因此我们等待少量重复的ACK到来。假如这只是一些报文段的重新排序，则在重新排序的报文段被处理并产生一个新的ACK之前，只可能产生1 ~ 2个重复的ACK。

如果一连串收到3个或3个以上的重复ACK，就非常可能是一个报文段丢失了。于是我们就重传丢失的数据报文段，而无需等待超时定时器溢出。这就是快速重传算法。接下来执行的不是慢启动算法而是拥塞避免算法。这就是快速恢复算法。

这个算法通常按如下过程进行实现：

1）当收到第3个重复的ACK时，将ssthresh设置为当前拥塞窗口cwnd的一半。重传丢失的报文段。设置cwnd为ssthresh加上3倍的报文段大小。

2）每次收到另一个重复的ACK时，cwnd增加1个报文段大小并发送1个分组（如果新的cwnd允许发送）。

3）当下一个确认新数据的ACK到达时，设置cwnd为ssthresh（在第1步中设置的值）。这个ACK应该是在进行重传后的一个往返时间内对步骤1中重传的确认。另外，这个ACK也应该是对丢失的分组和收到的第1个重复的A C K之间的所有中间报文段的确认。这一步采用的是拥塞避免，因为当分组丢失时我们将当前的速率减半。

六、代理 socks5 支持

参照 RFC1928、RFC1929，在 TDP 实现中，支持匿名通过 socks5 代理以及用户名/密码验证方式通过 socks5 代理。

由于 socks5 代理是工作于运输层上，因此连接当中对 IP 层选项的设置都将没有效果。socks5 代理起到的作用只是应用数据的转发，但这已经基本上能支持大部分用户程序的应用需求。在使用 socks5 代理进行工作中，路径 MTU 的发现机制，将无法有效工作，此时 MSS 默认为 536（MTU 默认为 576）,用户程序可以修改使用的 MSS 值。

七、安全考虑

TDP 协议及算法方面并不对数据的安全性做任何考虑，用户程序在传输数据时如果对安全性有要求,可以自行在应用数据层做相应的工作。但TDP 实现中,会提供一个简单的AES256 位加解密方法，提供给用户程序使用。用户程序可以调用该加解密方法，对数据进行加密然后再通过网络进行发送,接收时将加密数据流进行解密再将会用户程序数据逻辑处理模块进行处理。

八、定时器

如 BSD 的 TCP 实现类似，TDP 也为每条连接建立了六个定时器，简要介绍如下：

1）“连接建立”定时器，在发送 SYN 报文段建立一条新的连接时启动。如果没有在 75 秒内收到响应，连接建立将中止。

2）“重传”定时器，在发送数据时设定。如果定时器已超时而对端的确认还未到达，将重传数据。重传定时器的值是动态计算的，取决来 RTT 与该报文段被重传的次数。

3）“延迟 ACK”定时器，收到必须确认但无需马上发出确认的数据时设定。等待 200ms 后发送确认响应。如果，在这 200ms 内，有数据要在该连接上发送，延迟的 ACK 响应就可随数据一起发送回对端，称为捎带确认。

4）“坚持”定时器，在连接对端通告接收窗口为 0,阻止继续发送数据时设定。坚持定时器在超时后向对端发送 1 字节的数据，判定对端接收窗口是否已经打开。坚持定时器的值是动态的计算的，取决于 RTT 值，在 5 秒与 60 秒之间取值。

5）“保活”定时器。TDP 连接在一定时间段内如果没有数据交互的话，将主动发送保活

LIV 报文段。即当“保活”定时器超时，说明没有数据交互，则发送保活数据包。保活定时器默认时间为 2 分钟，用户程序可以进行设置。

6）TIME_WAIT 定时器,也可称为 2MSL 定时器（实现中，一个 MSL 为 1 分钟）。当连接状态转移到 TIME_WAIT 时，即连接主动关闭时，定时器启动。

九、开发接口

使用 TDP 进行网络程序开发是非常容易的，它的开发接口（API）与 socket API 是非常相似的，尤其是对应功能的函数名称都是一致的，需要注意的是 TDP 的所有 API 都处于名称空间 TDP 之下。开发接口见下表：

函数	描述
TDP::accept	接受一个链接
TDP::bind	绑定本地地址到一个 TDP::SOCKET 句柄
TDP::cleanup	清除 TDP 全局资源，一个进程中只需要调用一次
TDP::close	关闭已打开的 TDP::SOCKET 句柄，并关闭连接
TDP::connect	连接到服务器端
TDP::getlasterror	获得 TDP 最后的一个错误
TDP::getpeername	读取连接的对端的地址信息
TDP::getsockname	读取连接的本地的地址信息
TDP::getsockopt	读取 TDP 的选项信息
TDP::listen	等待客户端来连接
TDP::recv	接收数据
TDP::select	等待集合中的 TDP SOCKET 改变状态
TDP::send	发送数据
TDP::setsockopt	修改 TDP 的选项信息
TDP::shutdown	指定关闭连接上双工通信的部分或全部
TDP::socket	创建一个 TDP SOCKET
TDP::startup	初始化 TDP 全局信息，一个进程中只需要调用一次

十、作者简介

黄洪波，男，1979 年 11 月 20 日出生；
2002.6 年毕业于江西农业大学计算机与信息工程学院；
2002.6-2003.7 年工作于先锋软件股份有限公司，从事电子政务及企业信息化研究与开发；
2003.7-2006.6 年工作于金山软件股份有限公司金山毒霸事业部，从事企业信息安全领域研发工作，专注于网络应用技术及安全的研究；
2006 年 6 月中旬至今，成为软件自由人，目前从事 TDP（TCP-over-UDP library）及项目 PlumBlossom 研发。