链滴

TCP 协议

作者：helly
原文链接：https://ld246.com/article/1511929084454
来源网站：链滴

TCP：可靠传输、按序到达。
一、TCP 报文段格式：
源端口号（2byte）、目的端口号（2byte）
序列号（4byte）
ACKnum（4byte）
Offset（4bit）、Reserved（4bit）、TCP Flags（1byte）、Window（2byt
）
Checksum（2byte）、...
...
二、三次握手：

客户端向服务器发送连接建立请求报文段（Segement）（SYN=1，seq=x）（调用了 connect(
方法），发送之后客户端进入 SYN_SEND 状态；
服务器收到客户端的请求报文段，向客户端发送一个 ACK 和连接建立请求结合在一起的报文段（
CK=1，ACKnum=x+1，SYN=1，seq=y），并从 LISTEN 状态（调用了 listen()方法）进入 SYN_R
VD 状态；
客户端收到 ACK 报文段后，向服务器发送一个 ACK 报文段（ACK=1，ACKnum=y+1），并进
ESTABLISHED 状态;
服务器收到 ACK 报文段，也进入 ESTABLISHED 状态。
至此，客户端和服务器的连接建立，之后就可以调用 read()和 write()方法进行读写操作了。

三、四次挥手：

客户端向服务器发送连接关闭请求报文段（FIN=1，seq=x）（调用了 close()方法），发送之后
户端进入 FIN_WAIT_1 状态；
服务器收到客户端的请求报文段，向客户端发送 ACK 报文段（ACK=1，ACKnum=x+1），并
入 CLOSE_WAIT 状态；
客户端收到服务器的 ACK 报文段后进入 FIN_WAIT_2 状态；
服务器向客户端发送连接关闭请求报文段（FIN=1，seq=y）（调用了 close()方法），发送之后
务器进入 LAST_ACK 状态；
客户端收到服务器的请求报文段，向服务器发送 ACK 报文段（ACK=1，ACKnum=y+1），并
入 TIME_WAIT 状态；
服务器收到客户端的 ACK 报文段，进入 CLOSE 状态；
客户端等待 2MSL，期间没有收到服务端的连接关闭请求博爱文段，则进入 CLOSE 状态。
至此，客户端与服务器的连接关闭。

MSL（Maxinum Segment Live）
TCP 报文在网络传输时最长生存时间。
客户端为什么要等待 2MSL？
因为 ACK 报文段可能在网络传输的过程中丢失。如果发送了这种情况，那么服务器在等待 2MSL 时
后会重新一个连接关闭请求报文段，客户端在等待的这 2MSL 时间内正好可以收到这个请求报文段，
是重发 ACK 报文段，再等待 2MSL，重复这个过程直到在等待的 2MSL 时间内没有收到请求报文段
进入 CLOSE 状态。
SYN Flood 攻击：
描述：给服务器发了一个 SYN 后，就下线了，于是服务器需要默认等 63s 才
断开连接，这样，攻击者就可以把服务器的 syn 连接的队列耗尽，让正常的连接请求不能处理。
解决方法：Linux 下给了一个叫 tcp_syncookies 的参数来应对这个事——当 S
N 队列满了后，TCP 会通过源地址端口、目标地址端口和时间戳打造出一个特别的 Sequence Numb
r 发回去（又叫 cookie），如果是攻击者则不会有响应，如果是正常连接，则会把这个 SYN Cookie
回来，然后服务端可以通过 cookie 建连接（即使你不在 SYN 队列中）。
三、确认机制：
一种是确认收到的最大的连续收到的包。
一种是收到一个包去人一个包。
四、超时重传机制：

一种是重传超时的包及其之后的包（因为接收端只确认了最大的连续收到的包），快，但浪费带宽。
br>
一种是仅重传超时的包，慢，但节省带宽。
往返时间（RTT，Round Trip Time）
通过 RTT+ 算法（好几种）算出超时重传时间（RTO，Retransmission TimeO
t）。
五、滑动窗口协议：
window：还有多少缓冲区可以用。
发送端：
LastByteAcked、LastByteSent、LastByteWriter
SendWindow、UsableWindow
接收器：
LastByteRead、NextByteExpected、LastByteRcvd
...、ReceiveWindow
window=MaxRcvBuffer - LastByteRcvd – 1
六、拥塞控制：
慢启动阈值（ssthresh，slow start threshold）
拥塞控制算法：
Tahoe：没有快速恢复阶段；
Reno：增加了快速恢复阶段。
1. 慢启动：
拥塞窗口（cwnd，Congestion Window）。
①. 连接建立后初始化 cwnd=1；
②. 每当收到一个 ACK，cwnd++，呈线性上升；
③. 每过一个 RTT，cwnd*=2，呈指数上升；
④. 当 cwnd>=慢启动阈值时，将 cwnd 减半，进入拥塞避免算法；
2. 拥塞避免：
①. 每收到一个 ACK，cwnd = cwnd + 1/cwnd；
②. 每过一个 RTT，cwnd += 1；
3. 快速重传：
收到 3 个重复 ACK 就重传包，然后 cwnd = cwnd /2，ssthresh = cwnd， 进入快速恢复阶段。<b
>
（以前是等到超过超时重传时间 RTO 再重传包，然后 ssthresh = cwnd /2，cwnd 置为 1，重新进
慢启动。）
4. 快速恢复：
快速恢复阶段认为，还有 3 个重复 ACK 说明网络并不那么糟糕，所以
①. cwnd = ssthresh + 3*MSS；
②. 重传重复 ACK 指定的包；
③. 如果再收到重复 ACK，直接 cwnd += 1；
④. 如果收到新的 ACK，那么 cwnd = ssthresh，然后进入拥塞避免算法。
这个快速恢复算法存在的问题就是它依赖于 3 个重复的 ACK。 注意，3 个重复的 ACK 并不代
只丢了一个数据包，很有可能是丢了好多包。但这个算法只会重传一个，而剩下的那些包只能等到 RT
超时，于是，进入了恶 梦模式——超时一个窗口就减半一下，多个超时会超成 TCP 的传输速度呈级
下降，而且也不会触发快速恢复算法了。