TCP

传输层和网络层协议

网络层

提供主机间的逻辑通信

提供尽力而为交付服务，但不做任何确保。例如，不能保证报文段的交付，不保证报文段的按序交付，不保证报文段中数据的完整性

传输层

在主机间逻辑通信之上，提供应用进程间逻辑通信

传输层最重要的TCP协议，最重要的功能就是在不可靠的网络层协议之上构建出可靠的、具有拥塞机制的交付服务

可靠数据传输原理

可靠交付
(串行发送与接收)

verson 1.0: 底层通信信道完全可靠，不会出现丢包、数据包损坏问题

接收方完全不需要给发送方任何反馈，因为不会出现任何的差错

version 2.0: 底层通信信道不会丢包，但会出现数据包损坏，例如某个bit损坏

接收方如何判断数据包是完整无损的？

发送方在发送之前添加数据校验和，接收方收到数据包后重新校验，并与数据校验和进行对比

接收方在判断数据包有损以后，如何告诉发送方？

校验成功回送OK，校验失败回送NOT OK

OK: Positive Acknowledgement - ACK
NOT OK: Nagative Acknowledgement - NAK

接收方回送NAK，即数据包在通信时发生了损坏，接收方应该怎么做？

发送方重新发送该报文

差错检测
接收方反馈
重传

version 2.0 有一个很重大的缺陷:没有考虑ACK或者NAK数据包受损的可能性

当发送方接收到受损的反馈包时，直接选择重传。

但这带来了一个新的问题: 假设接收方回送的ACK受损，发送方收到损坏的ACK之后决定重传，上一个数据包将重新被发送，抵达接收方时，此时接收方并不能知道这是一个新的数据包还是发送方重传的数据包!

所以需要在数据包中添加序列号标志，如此一来接收方就能知道是新的报文还是重传的报文了

序列号

version 3.0: 底层通信不仅会产生比特损坏，还可能会丢包

当数据包在网络通信过程中丢失后，接收方完全不知道有该数据包的发送，也无法回送NAK

既然发送方知道自己发送了某个包，那么在一段时间后没有收到接收方反馈的数据包，大概率能够证明当前数据包已丢失，发送方将重传该数据包

所以，发送方在发送一个数据包以后即开启一个定时器，当定时器到期以后仍未收到接收方反馈，则进行重传

定时器

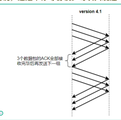
version 3.0综合使用差错检测(校验和)，接收方反馈(ACK与NAK)，数据重传，序列号以及定时器，得到了一个可靠数据传输协议

流水线可靠数据传输协议

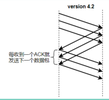
在version 3.0中，虽然使用各种手段实现了一个可靠传输协议，但是它是基于串行发送与串行接收为前提的，换句话说，这是个停等协议。每次只发送一个数据包，收到数据包的ACK之后再发下一个，在效率上不尽人意

每次发送多个(N个)数据包

version 4.1: 这N个数据包的反馈全部收到，并且是ACK时，发送下一组N个数据包



version 4.2: 每收到一个数据包的ACK反馈，就发送一个新的数据包



如此一来，除了发送方需要添加序列号以外，反馈的ACK或者NAK包也同样需要添加序列号。并且需要增加序号大小范围

发送方和接收方必须缓存多个数据包，发送方至少需要缓存已发送但未确认的数据包，接收方至少缓存一部分已经接收并确认的数据包

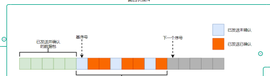
需要的序列号范围以及对缓存的要求取决于如何处理丢失、损坏或者超时的数据包

流水线差错恢复处理方式

只有基序号的确认包正确返回以后，下一个序号的数据包才会发送

窗口长度表示已发送但未确认的数据包最大数量，随着数据包的发送与ACK的接收，窗口向后滑动，因此常常被称为滑动窗口协议

回退N步



尽管在窗口内接收了5个数据包的确认，但是窗口仍然不能向右滑动: 因为基序号仍未确认

滑动窗口必须满足的一个规则: 窗口左侧必须是已发送且正确接收的数据包

当基序号的定时器超时时，协议会将窗口内的所有数据包进行再次发送，这就是回退N步的含义: 我不在乎谁成功了谁失败了，只要有失败的，我就将窗口内最开始失败的地方重新发送

选择重传　子主题 1

TCP协议

TCP连接

TCP连接只是逻辑上的概念，并不是一条端到端的电路或者是虚电路，其连接状态完全保存在两个端系统中。也就是说，将连接的状态进行删除，那么这条所谓的TCP连接也就不复存在了

TCP的报文并不会直接发送给网卡，而是先进入TCP发送缓存，而后再发送至网卡。接受方收到报文以后，也不会直接发送给应用程序，而是发送至TCP接收缓存，由内核发送至应用程序

首先，缓存报文的目的在于可靠数据传输需要，当出现丢包或者是损坏时需要重传

缓冲池的存在可以使得应用程序不被阻塞，例如非阻塞套接字

便于TCP拥塞机制的建立

TCP报文结构

简要分析

传输层提供进程间通信，而在网络通信中，进程的PID不能作为进程标识，而使用端口号代替，所以，TCP的报文结构中一定会有源端口号以及目的端口号

上面的可靠传输原理中，为了应对数据包的丢失以及能够使用流水线的方式发送数据包，那么必须要有校验和和序列号

此外，接收方在收到相应的数据包以后，需要向发送方确认，回送ACK或者是NAK，那么TCP报文中应当也有该字段

实际结构

32位的序列号与确认号

序列号

TCP将数据看成是无结构的有序字节流，所以其序列号是建立在传送的字节流之上，而不是建立在传送的报文段序列之上

比如2000个字节，每次传输1000个，若第一个报文序列号为55的话，第二个报文的序列号将会是1000，而不是1

实际上，TCP的起始序列号是一个随机数

确认号

发送方期望从接收方收到的下一字节的序号

比如主机A收到了来自主机B的序列号为805的报文，数据长度为50，A下一次想要的序列号是多少? 当前序列号+传输的数据长度，即确认号应为855

假设甲发送了"Seq: x Len: y"的数据给乙，那么乙的确认号为x+y，表示它已经收到了x+y之前的所有字节

16位的接收窗口　用于流量控制

PSH、URG　暂未使用

ACK　1字节确认标识，用来指示确认字段中的值是有效的，成功接收报文的反馈信息

RST、SYN、FIN　用于连接的建立与拆除

TCP连接的建立(三次握手)

TCP的连接仅仅是逻辑上的连接，并不存在实际物理链路上的连接。在连接的过程，双方需要确定对方是否能够应答，以及确定传输初期的窗口大小

SYN标识位　携带这个标识的包表示正在发起连接请求。因为连接是双向的，所以建立连接时，双方都需要发一个SYN

Source	Destination	Protocol	Length	Info
192.168.1.112	172.217.160.106	TCP	74	42110 → 443 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1 TSval=33935…
172.217.160.106	192.168.1.112	TCP	74	443 → 42110 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1350 SACK_PERM=1…
192.168.1.112	172.217.160.106	TCP	66	42110 → 443 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=3393554099 TSecr=339…

Wireshark开启了Relative Sequence Number，否则其实Seq一般不为0

112发起TCP连接，发送SYN标识包，初始化序列号为X, 并告知对方自己的窗口大小(Win)为64240，最大分段大小(MSS)为1460

106收到SYN包，即连接请求后进行应答: 初始化序列号为Y，Ack=X+1，窗口大小为64240，MSS为1350

112收到106的SYN包以后，发送确认包，此时Seq=X+1，Ack=Y+1, 窗口大小为64256

MSS

网络对数据包的大小是有限制的，不同的物理网络限制大小不同，最大值称为MTU，即最大传输单元。大多数网络的MTU为1500字节，除去IP头(20字节)与TCP头(20字节)后，能够传输的最大数据长度为1460字节

客户端和服务端通常处于不同的网络环境下，MTU也不尽相同，所以，需要在连接建立时告知对方自己的MSS大小。通过该值，发送放决定一次到底传输多大的数据

假设服务端的MSS为8960，客户端的MSS大小为1460，那么客户端向服务端发送数据时，会一次性发送8960字节的数据吗?

不会，因为客户端的MTU就只有1500(1460+20+20)，封包过大直接会被客户端的网络所丢弃，根本发不到服务端中

如同木桶理论一样，决定每次传输的MSS大小的，是MSS较小的那一方

Win

窗口大小即为接收方能够积累的数据总量，例如上图中的64240，表示接收方能够积累(缓存)最多64240字节的数据

窗口大小和MSS的大小可以用一个生活中的例子进行举例

你在盒马上买包子，家里面冰箱最多能装下20包包子，假设快递员每次只能送10包，那么他得分两次配送。若每次能送30包，他也不会送30包过来，因为冰箱就只能装20包，啥时候吃完了，啥时候再送

冰箱容量就是窗口大小

快递员小哥的运力就是MSS大小

MSS和Win的最小值，就是TCP一次能够发送的最大数据长度

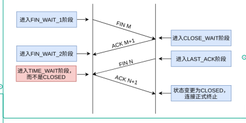
添加在TCP报文中的窗口大小是声明发送方能够接收的数据窗口大小，并不是表示发送窗口大小

事实上，发送窗口的大小在实际网络环境中很难确定

在三次握手以后，双方确认了对方的起始序列号以及窗口大小、MSS大小，这些信息就是TCP逻辑上的连接

TCP连接的拆除(四次挥手)

TCP连接的拆除要比连接建立更为复杂一些，其原因在于需要实现可靠地全双工连接的终止



当连接的一方想要关闭连接时，就向对方发起带有FIN标识的报文，接收方收到FIN报文后，回送ACK

过了一会，另一端也准备关闭连接，发送FIN报文，接收方在接收到此FIN报文后，并不会立即关闭当前TCP连接，而是进入TIME_WAIT阶段，等一会在完全释放相关资源，并回送ACK

为什么需要TIME_WAIT，而不是直接关闭连接
(假设左侧为客户端，右侧为服务端)

确保连接终止

假设客户端发送给服务端的ACK N+1丢失，且客户端完全关闭连接，释放所有资源。那么服务端ACK确认超时时，会重新发送FIN。

客户端终止后相同端口没有被其它进程使用，那么服务端永远收不到FIN的ACK，只能做无用的尝试然后达到最大超时次数关闭

客户端终止后相同端口被其它进程使用，并且已经建立好了TCP连接，这时候出现了一个FIN包，客户端该怎么处理? 很明显，客户端绝对不会回送ACK

确保原来的TCP报文不会出现在新建立的TCP连接中

这就是上面所述的，防止FIN报文出现在了后续端口对应的进程中而造成奇怪的影响