TCP/UDP协议（传输层）

- TCP基本认识
  - TCP头格式

    序列号：在建立连接时由计算机生成的随机数作为其初始值，用来解决网络包乱序问题。例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100 字节，那么下一个报文段的序号应为 401。 确认号：用来解决不丢包的问题。期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为 501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。 控制位： ACK：该位为 1 时，「确认应答」的字段变为有效，TCP 规定除了最初建立连接时的 SYN 包之外该位必须设置为 1 。 RST：该位为 1 时，表示 TCP 连接中出现异常必须强制断开连接。 SYC：该位为 1 时，表示希望建立连，并在其「序列号」的字段进行序列号初始值的设定。 FIN：该位为 1 时，表示今后不会再有数据发送，希望断开连接。当通信结束希望断开连接时，通信双方的主机之间就可以相互交换 FIN 位置为 1 的 TCP 段。
  - UDP头格式

    目标和源端口：主要是告诉 UDP 协议应该把报文发给哪个进程。 包长度：该字段保存了 UDP 首部的长度跟数据的长度之和。 校验和：校验和是为了提供可靠的 UDP 首部和数据而设计
  - 什么是 TCP（传输控制协议）？

    TCP 是面向连接的、可靠的、基于字节流的传输层通信协议。
    - 面向连接

      一定是「一对一」才能连接
    - 可靠的

      无论的网络链路中出现了怎样的链路变化，TCP 都可以保证一个报文一定能够到达接收端；
    - 字节流

      消息是「没有边界」的，所以无论我们消息有多大都可以进行传输。并且消息是「有序的」，当「前一个」消息没有收到的时候，即使它先收到了后面的字节已经收到，那么也不能扔给应用层去处理，同时对「重复」的报文会自动丢弃。
  - 什么是 UDP协议（用户数据报协议）？

    UDP是无连接，不可靠，面向报文的用户数据协议
    - 无连接
    - 不可靠
    - 面向报文
  - 为什么需要 TCP 协议？TCP 工作在哪一层?

    IP 层是「不可靠」的，它不保证网络包的交付、不保证网络包的按序交付、也不保证网络包中的数据的完整性。 如果需要保障网络数据包的可靠性，那么就需要由上层（传输层）的 TCP 协议来负责。 因为 TCP 是一个工作在传输层的可靠数据传输的服务，它能确保接收端接收的网络包是无损坏、无间隔、非冗余和按序的。
  - 如何唯一确定一个 TCP 连接呢

    TCP 四元组可以唯一的确定一个连接，四元组包括如下： 源地址 源端口 目的地址 目的端口
    - 源地址
    - 源端口
    - 目的地址
    - 目的端口
  - 一个端口的 TCP 的最大连接数是多少?

    最大TCP连接数 = 客户端IP数 * 客户端端口数
    - 客户端的 IP 最大数量?

      2 的 32 次方
    - 客户端端口最大数量?

      2 的 48 次方
  - UDP 和 TCP
    - 区别
      - 连接

        TCP 是面向连接的传输层协议，传输数据前先要建立连接。 UDP 是不需要连接，即刻传输数据。
      - 服务对象

        TCP 是一对一的两点服务，即一条连接只有两个端点。 UDP 支持一对一、一对多、多对多的交互通信
      - 可靠性

        TCP 是可靠交付数据的，数据可以无差错、不丢失、不重复、按需到达。 UDP 是尽最大努力交付，不保证可靠交付数据。
      - 首部开销

        TCP 首部长度较长，会有一定的开销，首部在没有使用「选项」字段时是 20 个字节，如果使用了「选项」字段则会变长的。 UDP 首部只有 8 个字节，并且是固定不变的，开销较小。
      - 拥塞控制、流量控制

        TCP 有拥塞控制和流量控制机制，保证数据传输的安全性。 UDP 则没有，即使网络非常拥堵了，也不会影响 UDP 的发送速率。
    - TCP应用场景
      - FTP 文件传输
      - HTTP / HTTPS
    - UDP应用场景
      - 广播通信
      - DNS 、SNMP
      - 视频、音频等多媒体通信
  - 为什么 UDP 头部没有「首部长度」字段，而 TCP 头部有「首部长度」字段呢?

    TCP 有可变长的「选项」字段，而 UDP 头部长度则是不会变化的，无需多一个字段去记录 UDP 的首部长度。
  - 为什么 UDP 头部有「包长度」字段，而 TCP 头部则没有「包长度」字段呢?

    为了网络设备硬件设计和处理方便，首部长度需要是 4字节的整数倍。如果去掉 UDP 「包长度」字段，那 UDP 首部长度就不是 4 字节的整数倍了
- TCP建立连接
  - TCP 三次握手过程和状态变迁
  - 为什么是三次握手?
    - 三次握手的好处
      - 避免历史连接

        客户端连续发送多次 SYN 建立连接的报文，在网络拥堵等情况下： 一个「旧 SYN 报文」比「最新的 SYN 」 报文早到达了服务端； 那么此时服务端就会回一个 SYN + ACK 报文给客户端； 客户端收到后可以根据自身的上下文，判断这是一个历史连接（序列号过期或超时），那么客户端就会发送 RST 报文给服务端，表示中止这一次连接。
      - 同步双方初始序列号

        当客户端发送携带「初始序列号」的 SYN 报文的时候，需要服务端回一个 ACK 应答报文，表示客户端的 SYN 报文已被服务端成功接收，那当服务端发送「初始序列号」给客户端的时候，依然也要得到客户端的应答回应，这样一来一回，才能确保双方的初始序列号能被可靠的同步。
      - 避免资源浪费

        如果只有「两次握手」，当客户端的 SYN 请求连接在网络中阻塞，客户端没有接收到 ACK 报文，就会重新发送 SYN ，由于没有第三次握手，服务器不清楚客户端是否收到了自己发送的建立连接的 ACK 确认信号，所以每收到一个 SYN 就只能先主动建立一个连接，这会造成什么情况呢？ 如果客户端的 SYN 阻塞了，重复发送多次 SYN 报文，那么服务器在收到请求后就会建立多个冗余的无效链接，造成不必要的资源浪费。
    - 两次握手
      - 无法防止历史连接的建立，会造成双方资源的浪费，也无法可靠的同步双方序列号；
    - 四次握手
      - 三次握手就已经理论上最少可靠连接建立，所以不需要使用更多的通信次数。
  - 序列号的作用?
    - 接收方可以去除重复的数据；
    - 接收方可以根据数据包的序列号按序接收；
    - 可以标识发送出去的数据包中， 哪些是已经被对方收到的；
  - 确认号的作用?
    - 保证数据包的完整性
    - 确认在这个序号以前的数据都已经被正常接收。
  - 第几次握手可以携带数据?

    第三次握手是可以携带数据的，前两次握手是不可以携带数据的
- TCP断开连接
  - TCP 四次挥手过程和状态变迁

    客户端打算关闭连接，此时会发送一个 TCP 首部 FIN 标志位被置为 1 的报文，也即 FIN 报文，之后客户端进入 FIN_WAIT_1 状态。 服务端收到该报文后，就向客户端发送 ACK 应答报文，接着服务端进入 CLOSED_WAIT 状态。 客户端收到服务端的 ACK 应答报文后，之后进入 FIN_WAIT_2 状态。 等待服务端处理完数据后，也向客户端发送 FIN 报文，之后服务端进入 LAST_ACK 状态。 客户端收到服务端的 FIN 报文后，回一个 ACK 应答报文，之后进入 TIME_WAIT 状态 服务器收到了 ACK 应答报文后，就进入了 CLOSE 状态，至此服务端已经完成连接的关闭。 客户端在经过 2MSL 一段时间后，自动进入 CLOSE 状态，至此客户端也完成连接的关闭。
  - 为什么挥手需要四次?

    TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出FIN报文段时， 只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据； 当主机2返回ACK报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的； 当主机2也发送了FIN报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1， 我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。
  - 为什么需要 TIME_WAIT 状态?
    - 防止旧连接的数据包

      如果主机1直接CLOSED，然后又再向主机2发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。 也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题， 但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中， 这些延迟数据在建立新连接之后才到达主机2，由于新连接和老连接的端口号是一样的，TCP协议就认为那个延迟的数据是属于新连接的， 这样就和真正的新连接的数据包发生混淆了。所以TCP连接还要在TIME_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失。
    - 保证TCP协议的全双工连接能够可靠关闭

      如果主机1直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致主机2没有收到主机1最后回复的ACK。 那么主机2就会在超时之后继续发送FIN，此时由于主机1已经CLOSED了，就找不到与重发的FIN对应的连接。 所以，主机1不是直接进入CLOSED，而是要保持TIME_WAIT，当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。
  - 为什么 TIME_WAIT 等待的时间是 2MSL?

    MSL是报文最大生存时间 被动关闭方没有收到断开连接的最后的 ACK 报文，就会触发超时重发 Fin 报文，另一方接收到 FIN 后，会重发 ACK 给被动关闭方， 一来一去正好 2 个 MSL 在 Linux 系统里 2MSL 默认是 60 秒，那么一个 MSL 也就是 30 秒
  - TIME_WAIT 过多有什么危害?
    - 内存资源占用
    - 对端口资源的占用，一个 TCP 连接至少消耗一个本地端口
- TCP粘包，拆包
  - 什么是粘包、拆包?

    粘包: 一个数据包中包含了发送端发送的两个数据包的信息 拆包：一个数据包被拆分成多次发送
  - 为什么会发生 TCP 粘包、拆包?

    要发送的数据大于 TCP 发送缓冲区剩余空间大小，将会发生拆包。 待发送数据大于 MSS（最大报文长度），TCP 在传输前将进行拆包。 要发送的数据小于 TCP 发送缓冲区的大小，TCP 将多次写入缓冲区的数据一次发送出去，将会发生粘包。 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包
  - TCP 有粘包和拆包
    - UDP 是基于报文发送的

      UDP首部采用了 16bit 来指示 UDP 数据报文的长度，因此在应用层能很好的将不同的数据报文区分开，从而避免粘包和拆包的问题
    - TCP 是基于字节流

      TCP 并没有把这些数据块区分边界，仅仅是一连串没有结构的字节流 在 TCP 的首部没有表示数据长度的字段
  - 粘包、拆包解决办法
    - 消息定长

      发送端将每个数据包封装为固定长度（不够的可以通过补 0 填充），这样接收端每次接收缓冲区中读取固定长度的数据就自然而然的把每个数据包拆分开来。
    - 设置消息边界

      服务端从网络流中按消息边界分离出消息内容。在包尾增加回车换行符进行分割，例如 FTP 协议。
    - 将消息分为消息头和消息体

      消息头中包含表示消息总长度（或者消息体长度）的字段
- TCP实现可靠性传输方式
  - 重传机制
    - 超时重传
      - 超时重传发生的条件
        - 数据包丢失
        - 确认应答丢失
      - 超时时间设置为多少呢?

        RTT 就是数据从网络一端传送到另一端所需的时间，也就是包的往返时间 超时重传时间 RTO 的值应该略大于报文往返 RTT 的值
      - 重传机制
        - 超时间隔加倍

          如果超时重发的数据，再次超时的时候，又需要重传的时候，TCP 的策略是超时间隔加倍。 也就是每当遇到一次超时重传的时候，都会将下一次超时时间间隔设为先前值的两倍。两次超时，就说明网络环境差，不宜频繁反复发送。
        - 不足：超时周期可能相对较长
    - 快速重传
      - 主旨：不以时间为驱动，而是以数据驱动重传
      - 重传机制

        发送方发出了 1，2，3，4，5 份数据： 第一份 Seq1 先送到了，于是就 Ack 回 2； 结果 Seq2 因为某些原因没收到，Seq3 到达了，于是还是 Ack 回 2； 后面的 Seq4 和 Seq5 都到了，但还是 Ack 回 2，因为 Seq2 还是没有收到； 发送端收到了三个 Ack = 2 的确认，知道了 Seq2 还没有收到，就会在定时器过期之前，重传丢失的 Seq2。 最后，收到了 Seq2，此时因为 Seq3，Seq4，Seq5 都收到了，于是 Ack 回 6 。 所以，快速重传的工作方式是当收到三个相同的 ACK 报文时，会在定时器过期之前，重传丢失的报文段。
      - 不足：重传的时候，是重传之前的一个，还是重传所有的问题
    - SACK
      - 主旨：只重传丢失的数据
      - 机制
    - D-SACK
      - 主旨：告诉「发送方」有哪些数据被重复接收了
      - 机制
        - ACK 丢包
        - 网络延时
      - 好处
        - 可以让「发送方」知道，是发出去的包丢了，还是接收方回应的 ACK 包丢了
        - 可以知道是不是「发送方」的数据包被网络延迟了
        - 可以知道网络中是不是把「发送方」的数据包给复制了
  - 滑动窗口
    - 作用：通信的效率
    - 窗口大小

      无需等待确认应答，而可以继续发送数据的最大值
    - 窗口大小由哪一方决定

      通常窗口的大小是由接收方的窗口大小来决定的
  - 流量控制

    TCP 提供一种机制可以让「发送方」根据「接收方」的实际接收能力控制发送的数据量
    - 作用：避免「发送方」的数据填满「接收方」的缓存
    - 机制

      客户端向服务端发送请求数据报文。这里要说明下，本次例子是把服务端作为发送方，所以没有画出服务端的接收窗口。 服务端收到请求报文后，发送确认报文和 80 字节的数据，于是可用窗口 Usable 减少为 120 字节，同时 SND.NXT 指针也向右偏移 80 字节后，指向 321，这意味着下次发送数据的时候，序列号是 321。 客户端收到 80 字节数据后，于是接收窗口往右移动 80 字节，RCV.NXT 也就指向 321，这意味着客户端期望的下一个报文的序列号是 321，接着发送确认报文给服务端。 服务端再次发送了 120 字节数据，于是可用窗口耗尽为 0，服务端无法再继续发送数据。 客户端收到 120 字节的数据后，于是接收窗口往右移动 120 字节，RCV.NXT 也就指向 441，接着发送确认报文给服务端。 服务端收到对 80 字节数据的确认报文后，SND.UNA 指针往右偏移后指向 321，于是可用窗口 Usable 增大到 80。 服务端收到对 120 字节数据的确认报文后，SND.UNA 指针往右偏移后指向 441，于是可用窗口 Usable 增大到 200。 服务端可以继续发送了，于是发送了 160 字节的数据后，SND.NXT 指向 601，于是可用窗口 Usable 减少到 40。 客户端收到 160 字节后，接收窗口往右移动了 160 字节，RCV.NXT 也就是指向了 601，接着发送确认报文给服务端。 服务端收到对 160 字节数据的确认报文后，发送窗口往右移动了 160 字节，于是 SND.UNA 指针偏移了 160 后指向 601，可用窗口 Usable 也就增大至了 200。
  - 拥塞控制
    - 作用：避免「发送方」的数据填满整个网络
    - 机制：
      - 慢启动

        当发送方每收到一个 ACK，拥塞窗口 cwnd 的大小就会加 1 当 cwnd < ssthresh 时，使用慢启动算法。 当 cwnd >= ssthresh 时，就会使用「拥塞避免算法」
      - 拥塞避免

        每当收到一个 ACK 时，cwnd 增加 1/cwnd 一直增长着后，网络就会慢慢进入了拥塞的状况了，于是就会出现丢包现象，这时就需要对丢失的数据包进行重传。 当触发了重传机制，也就进入了「拥塞发生算法」
      - 拥塞发生
      - 快速恢复

        拥塞窗口 cwnd = ssthresh + 3 （3 的意思是确认有 3 个数据包被收到了）； 重传丢失的数据包； 如果再收到重复的 ACK，那么 cwnd 增加 1； 如果收到新数据的 ACK 后，把 cwnd 设置为第一步中的 ssthresh 的值，原因是该 ACK 确认了新的数据，说明从 duplicated ACK 时的数据都已收到，该恢复过程已经结束，可以回到恢复之前的状态了，也即再次进入拥塞避免状态