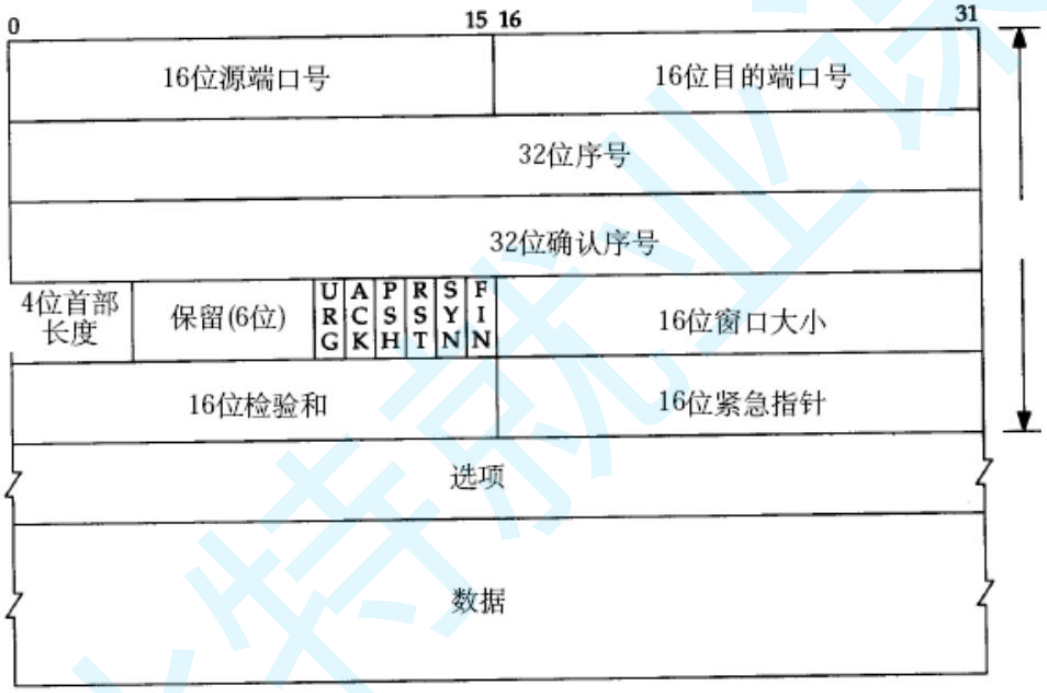
7 传输层协议 TCP

TCP 协议

TCP 全称为 "传输控制协议(Transmission Control Protocol"). 人如其名, 要对数据的传输进行一个详细的控制;

TCP 协议段格式



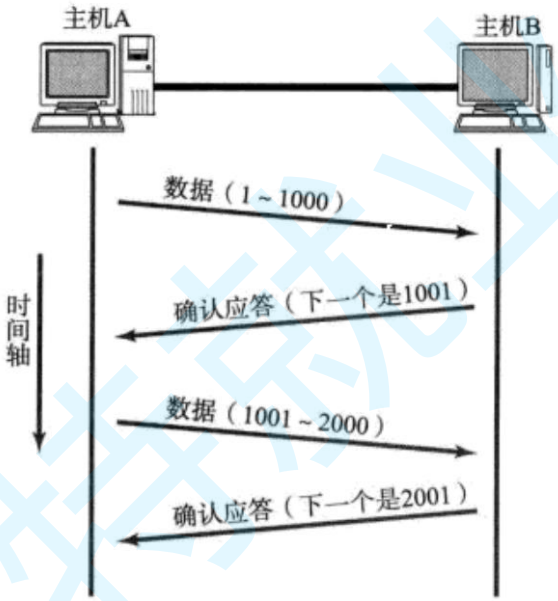
- 源/目的端口号: 表示数据是从哪个进程来, 到哪个进程去;
- 32 位序号/32 位确认号: 后面详细讲;
- 4 位 TCP 报头长度: 表示该 TCP 头部有多少个 32 位 bit(有多少个 4 字节); 所以 TCP 头部最大长度是 15 * 4 = 60
- 6 位标志位:
  - URG: 紧急指针是否有效
  - ACK: 确认号是否有效
  - PSH: 提示接收端应用程序立刻从 TCP 缓冲区把数据读走
  - RST: 对方要求重新建立连接; 我们把携带 RST 标识的称为**复位报文段**

- SYN: 请求建立连接; 我们把携带 SYN 标识的称为**同步报文段**
- FIN: 通知对方, 本端要关闭了, 我们称携带 FIN 标识的为**结束报文段**

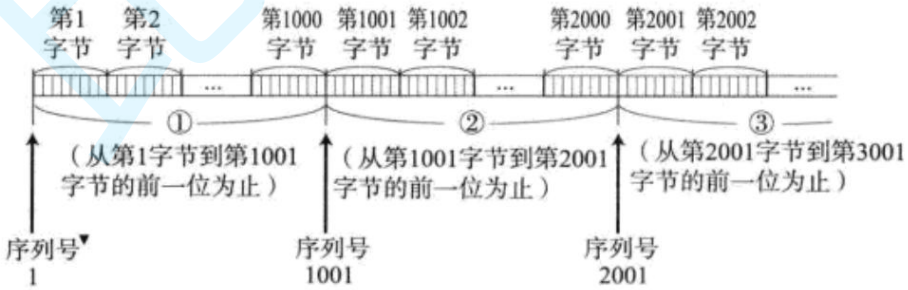
- 16 位窗口大小: 后面再说
- 16 位校验和: 发送端填充, CRC 校验. 接收端校验不通过, 则认为数据有问题. 此处的检验和不光包含 TCP 首部, 也包含 TCP 数据部分.
- 16 位紧急指针: 标识哪部分数据是紧急数据;
- 40 字节头部选项: 暂时忽略;

确认应答(ACK)机制

[唐僧讲经例子]

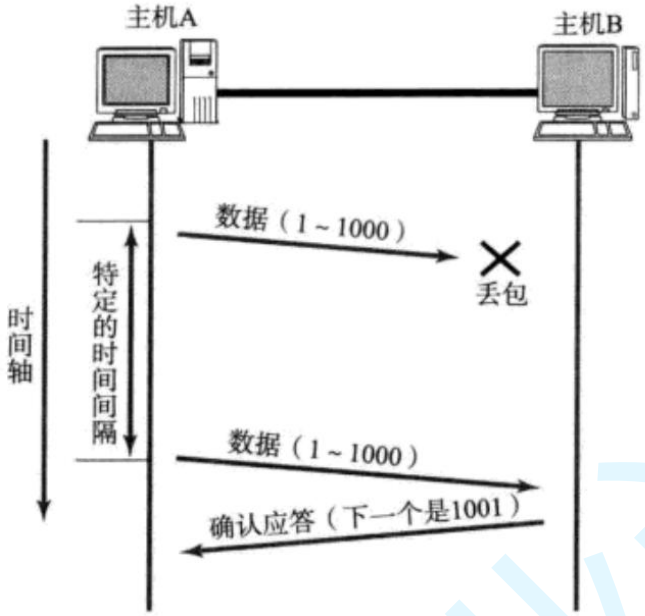


TCP 将每个字节的数据都进行了编号. 即为序列号.



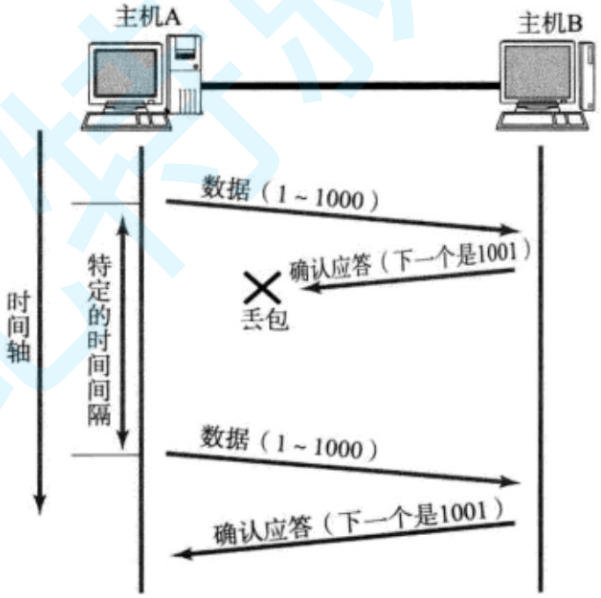
每一个 ACK 都带有对应的确认序列号, 意思是告诉发送者, 我已经收到了哪些数据; 下一次你从哪里开始发.

超时重传机制



- 主机 A 发送数据给 B 之后, 可能因为网络拥堵等原因, 数据无法到达主机 B;
- 如果主机 A 在一个特定时间间隔内没有收到 B 发来的确认应答, 就会进行重发;

但是, 主机 A 未收到 B 发来的确认应答, 也可能是因为 ACK 丢失了;



因此主机 B 会收到很多重复数据. 那么 TCP 协议需要能够识别出那些包是重复的包, 并且把重复的丢弃掉.

这时候我们可以利用前面提到的序列号, 就可以很容易做到去重的效果.

那么, 如果超时的时间如何确定?

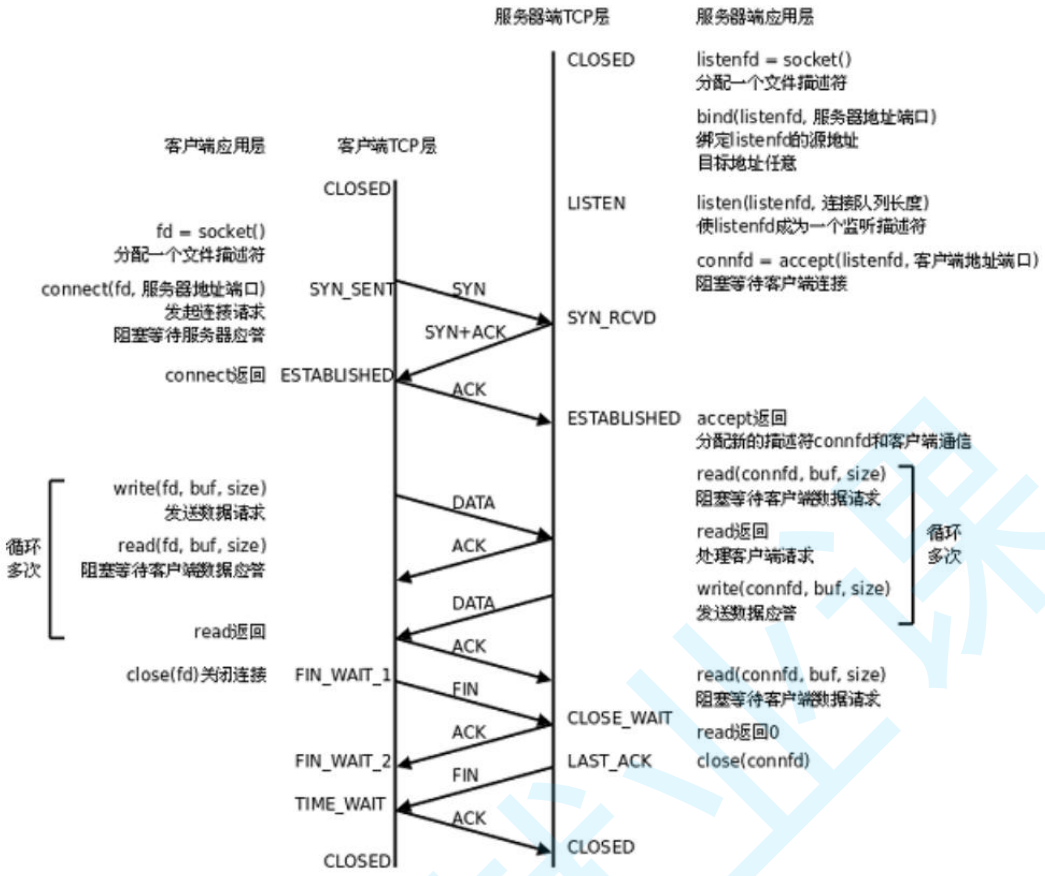
- 最理想的情况下, 找到一个最小的时间, 保证 "确认应答一定能在这个时间内返回".
- 但是这个时间的长短, 随着网络环境的不同, 是有差异的.
- 如果超时时间设的太长, 会影响整体的重传效率;
- 如果超时时间设的太短, 有可能会频繁发送重复的包;

TCP 为了保证无论在任何环境下都能比较高性能的通信, 因此会动态计算这个最大超时时间.

- Linux 中(BSD Unix 和 Windows 也是如此), 超时以 500ms 为一个单位进行控制, 每次判定超时重发的超时时间都是 500ms 的整数倍.
- 如果重发一次之后, 仍然得不到应答, 等待 2*500ms 后再进行重传.
- 如果仍然得不到应答, 等待 4*500ms 进行重传. 依次类推, 以指数形式递增.
- 累计到一定的重传次数, TCP 认为网络或者对端主机出现异常, 强制关闭连接.

连接管理机制

在正常情况下, TCP 要经过三次握手建立连接, 四次挥手断开连接



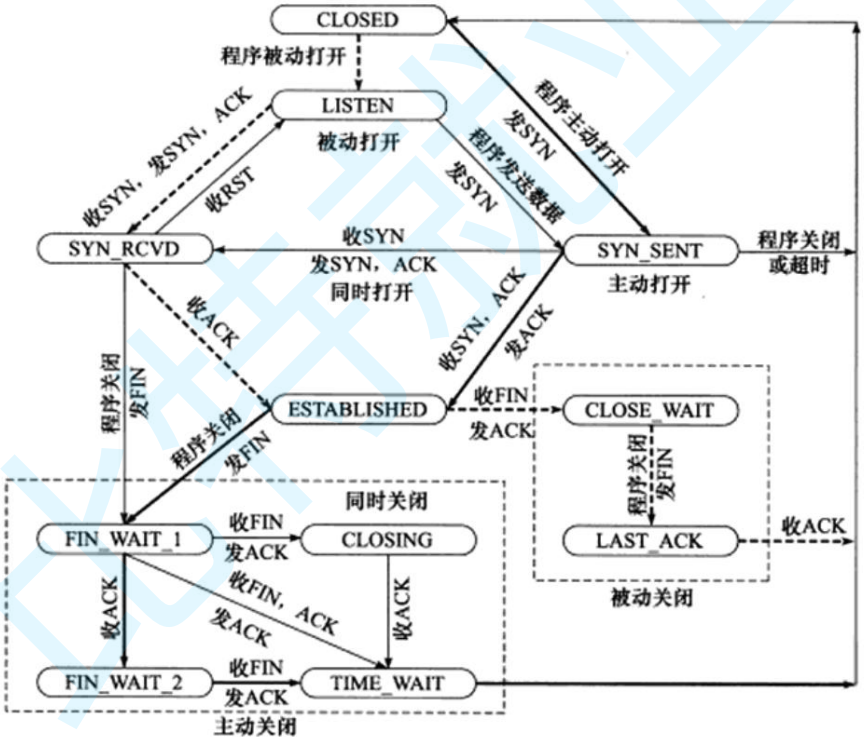
服务端状态转化:

- [CLOSED -> LISTEN] 服务器端调用 listen 后进入 LISTEN 状态, 等待客户端连接;
- [LISTEN -> SYN_RCVD] 一旦监听到连接请求(同步报文段), 就将该连接放入内核等待队列中, 并向客户端发送 SYN 确认报文.
- [SYN_RCVD -> ESTABLISHED] 服务端一旦收到客户端的确认报文, 就进入 ESTABLISHED 状态, 可以进行读写数据了.
- [ESTABLISHED -> CLOSE_WAIT] 当客户端主动关闭连接(调用 close), 服务器会收到结束报文段, 服务器返回确认报文段并进入 CLOSE_WAIT;
- [CLOSE_WAIT -> LAST_ACK] 进入 CLOSE_WAIT 后说明服务器准备关闭连接(需要处理完之前的数据); 当服务器真正调用 close 关闭连接时, 会向客户端发送 FIN, 此时服务器进入 LAST_ACK 状态, 等待最后一个 ACK 到来(这个 ACK 是客户端确认收到了 FIN)
- [LAST_ACK -> CLOSED] 服务器收到了对 FIN 的 ACK, 彻底关闭连接.

客户端状态转化:

- [CLOSED -> SYN_SENT] 客户端调用 connect, 发送同步报文段;
- [SYN_SENT -> ESTABLISHED] connect 调用成功, 则进入 ESTABLISHED 状态, 开始读写数据;
- [ESTABLISHED -> FIN_WAIT_1] 客户端主动调用 close 时, 向服务器发送结束报文段, 同时进入 FIN_WAIT_1;
- [FIN_WAIT_1 -> FIN_WAIT_2] 客户端收到服务器对结束报文段的确认, 则进入 FIN_WAIT_2, 开始等待服务器的结束报文段;
- [FIN_WAIT_2 -> TIME_WAIT] 客户端收到服务器发来的结束报文段, 进入 TIME_WAIT, 并发出 LAST_ACK;
- [TIME_WAIT -> CLOSED] 客户端要等待一个 2MSL(Max Segment Life, 报文最大生存时间)的时间, 才会进入 CLOSED 状态.

下图是 TCP 状态转换的一个汇总:



- 较粗的虚线表示服务端的状态变化情况;
- 较粗的实线表示客户端的状态变化情况;
- CLOSED 是一个假想的起始点, 不是真实状态;

关于 "半关闭" , **男女朋友分手例子**

关于 CLOSING 状态. 同学们可以课后调研一下.

理解 TIME_WAIT 状态

现在做一个测试,首先启动 server,然后启动 client,然后用 Ctrl-C 使 server 终止,这时马上再运行 server, 结果是:

```
$ ./server
 bind error: Address already in use
```

这是因为,虽然 server 的应用程序终止了,但 TCP 协议层的连接并没有完全断开,因此不能再次监 听同样的 server 端口. 我们用 netstat 命令查看一下:

```
$ netstat -apn |grep 8000
 tcp        1      0 127.0.0.1:33498         127.0.0.1:8000
CLOSE_WAIT 10830/client
 tcp        0      0 127.0.0.1:8000          127.0.0.1:33498
FIN_WAIT2  -
```

- TCP 协议规定,主动关闭连接的一方要处于 TIME_ WAIT 状态,等待两个 MSL(maximum segment lifetime)的时间后才能回到 CLOSED 状态.
- 我们使用 Ctrl-C 终止了 server, 所以 server 是主动关闭连接的一方, 在 TIME_WAIT 期间仍然不能再次监听同样的 server 端口;
- MSL 在 RFC1122 中规定为两分钟,但是各操作系统的实现不同, 在 Centos7 上默认配置的值是 60s;
- 可以通过 `cat /proc/sys/net/ipv4/tcp_fin_timeout` 查看 msl 的值;
- 规定 TIME_WAIT 的时间请读者参考 UNP 2.7 节;

```
[tangzhong@tz http]$ cat /proc/sys/net/ipv4/tcp_fin_timeout
60
```

想一想, 为什么是 TIME_WAIT 的时间是 2MSL?

- MSL 是 TCP 报文的最大生存时间, 因此 TIME_WAIT 持续存在 2MSL 的话
- 就能保证在两个传输方向上的尚未被接收或迟到的报文段都已经消失(否则服务器立刻重启, 可能会收到来自上一个进程的迟到的数据, 但是这种数据很可能是错误的);
- 同时也是在理论上保证最后一个报文可靠到达(假设最后一个 ACK 丢失, 那么服务器会再重发一个 FIN. 这时虽然客户端的进程不在了, 但是 TCP 连接还在, 仍然可以重发 LAST_ACK);

解决 TIME_WAIT 状态引起的 bind 失败的方法(作业)

在 server 的 TCP 连接没有完全断开之前不允许重新监听, 某些情况下可能是不合理的

> - 服务器需要处理非常大量的客户端的连接(每个连接的生存时间可能很短, 但是每秒都有很大数量的客户端来请求).
> - 这个时候如果由服务器端主动关闭连接(比如某些客户端不活跃, 就需要被服务器端主动清理掉), 就会产生大量 TIME_WAIT 连接.
> - 由于我们的请求量很大, 就可能导致 TIME_WAIT 的连接数很多, 每个连接都会占用一个通信五元组(源 ip, 源端口, 目的 ip, 目的端口, 协议). 其中服务器的 ip 和端口和协议是固定的. 如果新来的客户端连接的 ip 和端口号和 TIME_WAIT 占用的链接重复了, 就会出现问题.

使用 setsockopt()设置 socket 描述符的 选项 SO_REUSEADDR 为 1, 表示允许创建端口号相同但 IP 地址不同的多个 socket 描述符

```
int opt = 1;
setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));
```

理解 CLOSE_WAIT 状态

以之前写过的 TCP 服务器为例, 我们稍加修改

将 `new_sock.Close();` 这个代码去掉.

```
C
#pragma once
#include <functional>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string& req, std::string*
resp)> Handler;

class TcpServer {
public:
  TcpServer(const std::string& ip, uint16_t port) : ip_(ip),
port_(port) {
  }

  bool Start(Handler handler) {
    // 1. 创建 socket;
    CHECK_RET(listen_sock_.Socket());
    // 2. 绑定端口号
    CHECK_RET(listen_sock_.Bind(ip_, port_));
```

```
    // 3. 进行监听
    CHECK_RET(listen_sock_.Listen(5));
    // 4. 进入事件循环
    for (;;) {
      // 5. 进行 accept
      TcpSocket new_sock;
      std::string ip;
      uint16_t port = 0;
      if (!listen_sock_.Accept(&new_sock, &ip, &port)) {
        continue;
      }
      printf("[client %s:%d] connect!\n", ip.c_str(), port);
      // 6. 进行循环读写
      for (;;) {
        std::string req;
        // 7. 读取请求. 读取失败则结束循环
        bool ret = new_sock.Recv(&req);
        if (!ret) {
          printf("[client %s:%d] disconnect!\n", ip.c_str(),
port);
          // [注意!] 将此处的关闭 socket 去掉
          // new_sock.Close();
          break;
        }
        // 8. 计算响应
        std::string resp;
        handler(req, &resp);
        // 9. 写回响应
        new_sock.Send(resp);
        printf("[%s:%d] req: %s, resp: %s\n", ip.c_str(), port,
               req.c_str(), resp.c_str());
      }
    }
    return true;
  }
private:
  TcpSocket listen_sock_;
  std::string ip_;
  uint64_t port_;
};
```

我们编译运行服务器. 启动客户端链接, 查看 TCP 状态, 客户端服务器都为 ESTABLELISHED 状态, 没有问题.

然后我们关闭客户端程序, 观察 TCP 状态

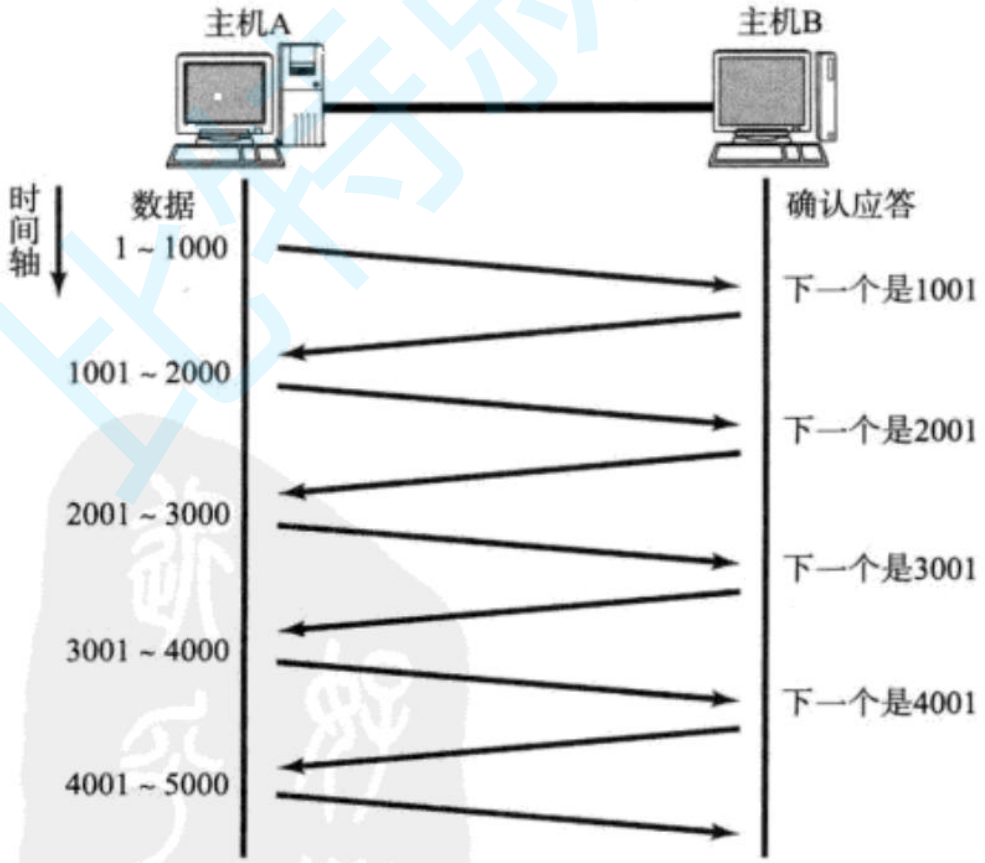
```
C
tcp        0      0 0.0.0.0:9090            0.0.0.0:*
LISTEN      5038/./dict_server
tcp        0      0 127.0.0.1:49958         127.0.0.1:9090
FIN_WAIT2   -
tcp        0      0 127.0.0.1:9090          127.0.0.1:49958
CLOSE_WAIT  5038/./dict_server
```

此时服务器进入了 CLOSE_WAIT 状态, 结合我们四次挥手的流程图, 可以认为四次挥手没有正确完成.

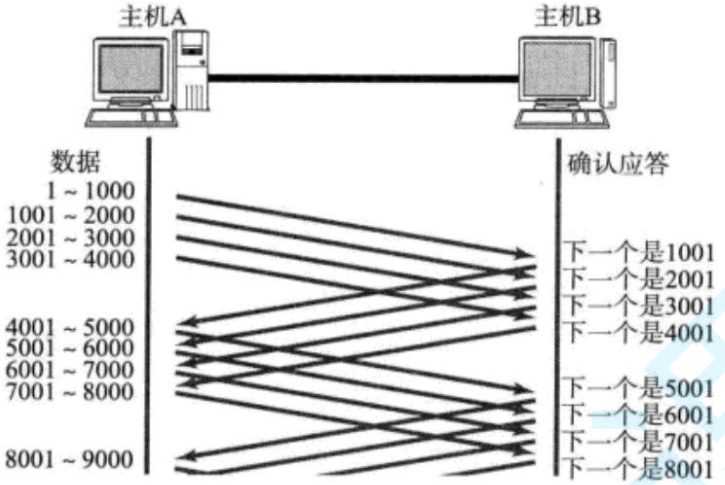
小结: 对于服务器上出现大量的 CLOSE_WAIT 状态, 原因就是服务器没有正确的关闭 socket, 导致四次挥手没有正确完成. 这是一个 BUG. 只需要加上对应的 close 即可解决问题.

滑动窗口

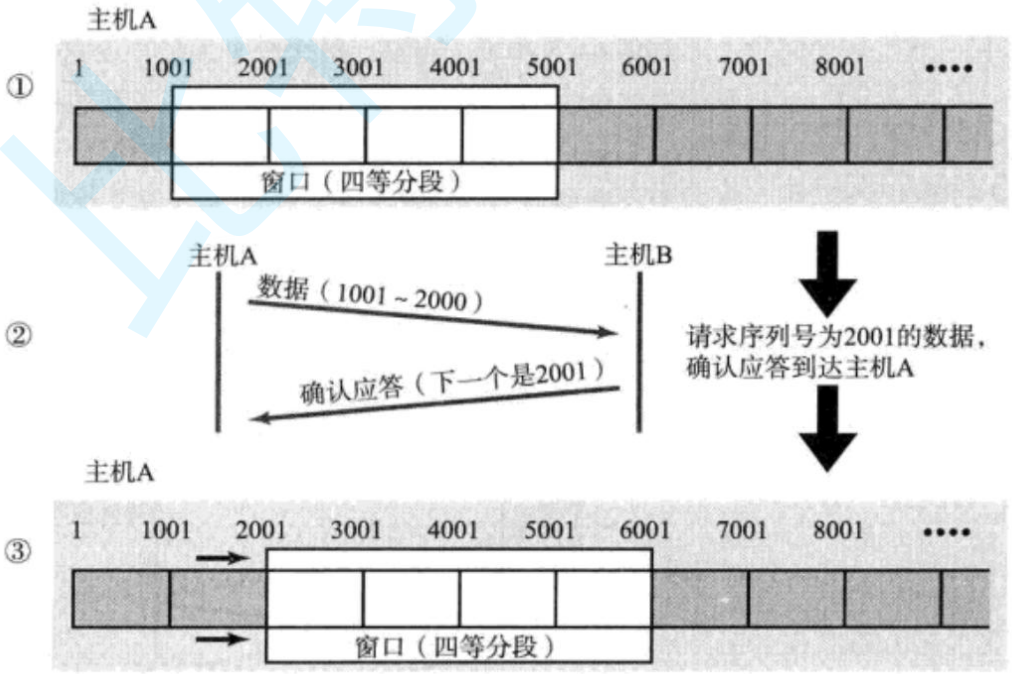
刚才我们讨论了确认应答策略, 对每一个发送的数据段, 都要给一个 ACK 确认应答. 收到 ACK 后再发送下一个数据段. 这样做有一个比较大的缺点, 就是性能较差. 尤其是数据往返的时间较长的时候.



既然这样一发一收的方式性能较低, 那么我们一次发送多条数据, 就可以大大的提高性能(其实是将多个段的等待时间重叠在一起了).

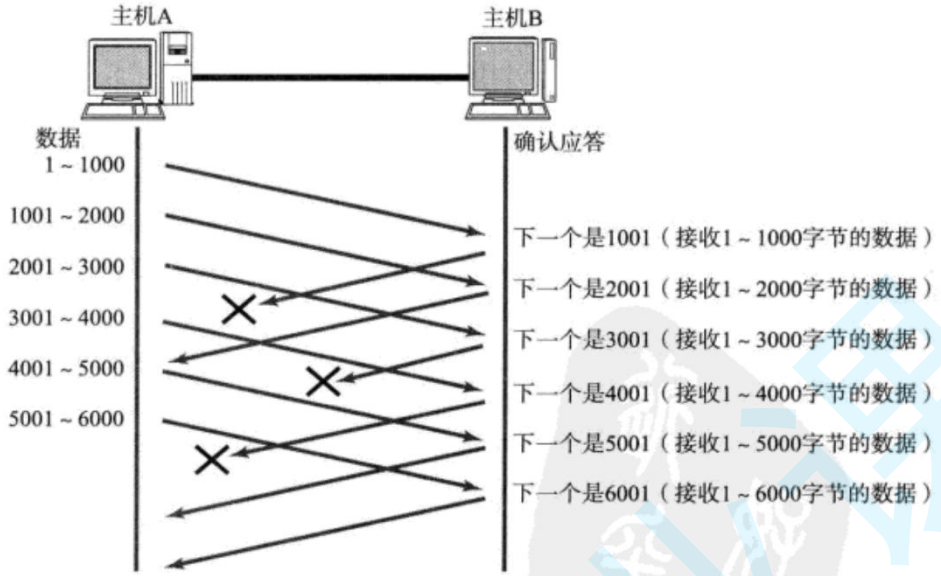


- 窗口大小指的是无需等待确认应答而可以继续发送数据的最大值. 上图的窗口大小就是 4000 个字节(四个段).
- 发送前四个段的时候, 不需要等待任何 ACK, 直接发送;
- 收到第一个 ACK 后, 滑动窗口向后移动, 继续发送第五个段的数据; 依次类推;
- 操作系统内核为了维护这个滑动窗口, 需要开辟 **发送缓冲区** 来记录当前还有哪些数据没有应答; 只有确认应答过的数据, 才能从缓冲区删掉;
- 窗口越大, 则网络的吞吐率就越高;



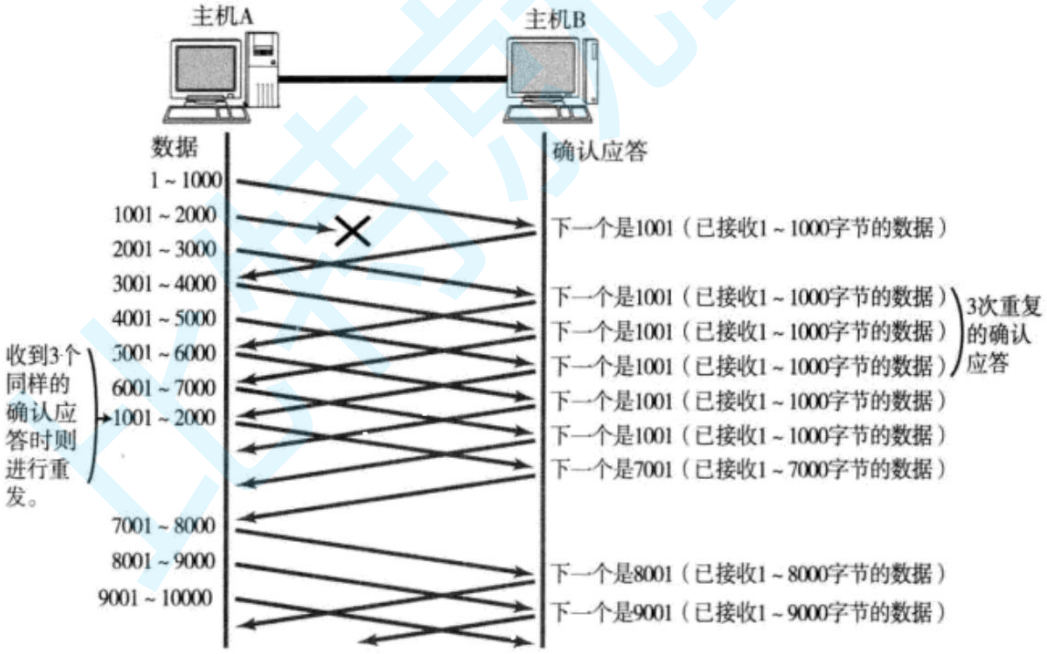
那么如果出现了丢包, 如何进行重传? 这里分两种情况讨论.

情况一: 数据包已经抵达, ACK 被丢了.



这种情况下, 部分 ACK 丢了并不要紧, 因为可以通过后续的 ACK 进行确认;

情况二: 数据包就直接丢了.



> - 当某一段报文段丢失之后, 发送端会一直收到 1001 这样的 ACK, 就像是在提醒发送端 "我想要的是 1001" 一样;
> - 如果发送端主机连续三次收到了同样一个 "1001" 这样的应答, 就会将对应的数据 1001 - 2000 重新发送;

- 这个时候接收端收到了 1001 之后, 再次返回的 ACK 就是 7001 了(因为 2001 - 7000)接收端其实之前就已经收到了, 被放到了接收端操作系统内核的**接收缓冲区**中;

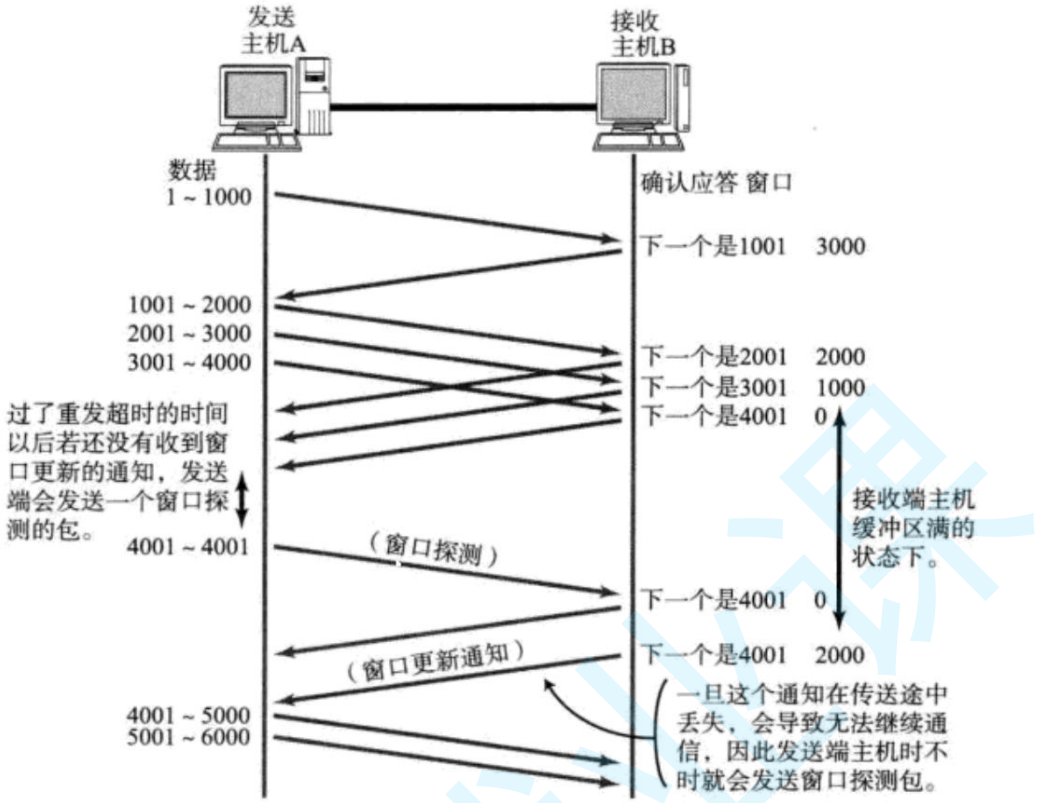
这种机制被称为 "高速重发控制"(也叫 "快重传").

流量控制

接收端处理数据的速度是有限的. 如果发送端发的太快, 导致接收端的缓冲区被打满, 这个时候如果发送端继续发送, 就会造成丢包, 继而引起丢包重传等等一系列连锁反应.

因此 TCP 支持根据接收端的处理能力, 来决定发送端的发送速度. 这个机制就叫做**流量控制(Flow Control)**;

- 接收端将自己可以接收的缓冲区大小放入 TCP 首部中的 "窗口大小" 字段, 通过 ACK 端通知发送端;
- 窗口大小字段越大, 说明网络的吞吐量越高;
- 接收端一旦发现自己的缓冲区快满了, 就会将窗口大小设置成一个更小的值通知给发送端;
- 发送端接受到这个窗口之后, 就会减慢自己的发送速度;
- 如果接收端缓冲区满了, 就会将窗口置为 0; 这时发送方不再发送数据, 但是需要定期发送一个窗口探测数据段, 使接收端把窗口大小告诉发送端.



接收端如何把窗口大小告诉发送端呢? 回忆我们的 TCP 首部中, 有一个 16 位窗口字段, 就是存放了窗口大小信息;

那么问题来了, 16 位数字最大表示 65535, 那么 TCP 窗口最大就是 65535 字节么?

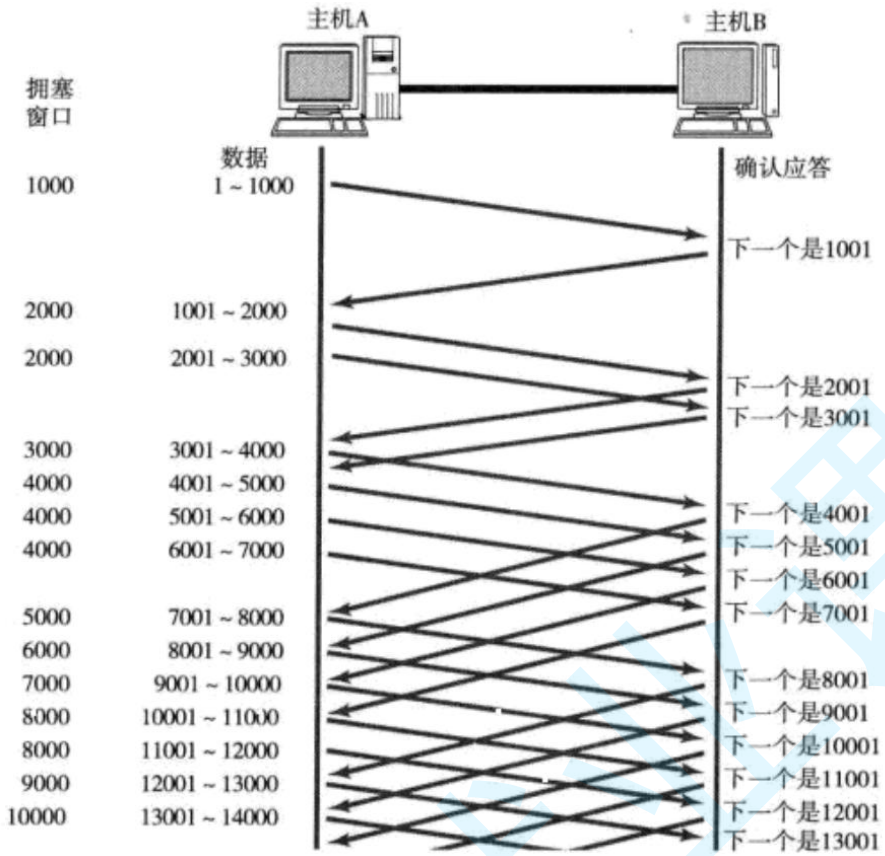
实际上, TCP 首部 40 字节选项中还包含了一个窗口扩大因子 M, 实际窗口大小是 窗口字段的值左移 M 位;

拥塞控制

虽然 TCP 有了滑动窗口这个大杀器, 能够高效可靠的发送大量的数据. 但是如果在刚开始阶段就发送大量的数据, 仍然可能引发问题.

因为网络上有很多的计算机, 可能当前的网络状态就已经比较拥堵. 在不清楚当前网络状态下, 贸然发送大量的数据, 是很有可能引起雪上加霜的.

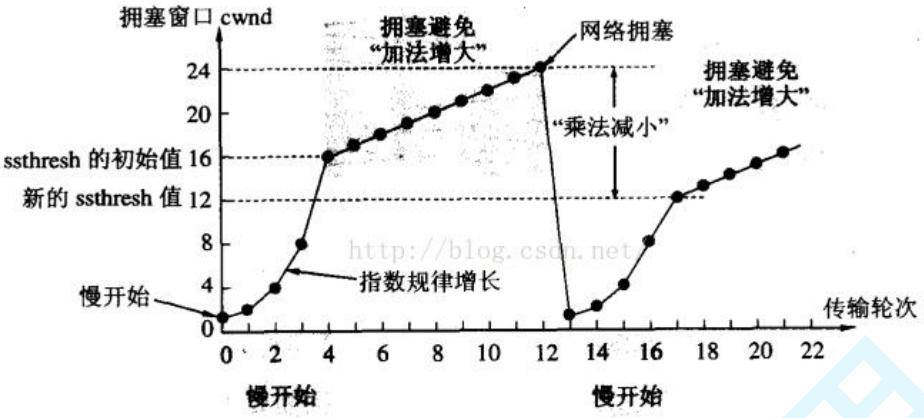
TCP 引入 **慢启动** 机制, 先发少量的数据, 探探路, 摸清当前的网络拥堵状态, 再决定按照多大的速度传输数据;



- 此处引入一个概念称为**拥塞窗口**
- 发送开始的时候, 定义拥塞窗口大小为 1;
- 每次收到一个 ACK 应答, 拥塞窗口加 1;
- 每次发送数据包的时候, 将拥塞窗口和接收端主机反馈的窗口大小做比较, 取较小的值作为实际发送的窗口;

像上面这样的拥塞窗口增长速度, 是指数级别的. "慢启动" 只是指初使时慢, 但是增长速度非常快.

- 为了不增长的那么快, 因此不能使拥塞窗口单纯的加倍.
- 此处引入一个叫做慢启动的阈值
- 当拥塞窗口超过这个阈值的时候, 不再按照指数方式增长, 而是按照线性方式增长



- 当 TCP 开始启动的时候, 慢启动阈值等于窗口最大值;
- 在每次超时重发的时候, 慢启动阈值会变成原来的一半, 同时拥塞窗口置回 1;

少量的丢包, 我们仅仅是触发超时重传; 大量的丢包, 我们就认为网络拥塞;

当 TCP 通信开始后, 网络吞吐量会逐渐上升; 随着网络发生拥堵, 吞吐量会立刻下降;

拥塞控制, 归根结底是 TCP 协议想尽可能快的把数据传输给对方, 但是又要避免给网络造成太大压力的折中方案.

TCP 拥塞控制这样的过程, 就好像 **热恋的感觉**

延迟应答

如果接收数据的主机立刻返回 ACK 应答, 这时候返回的窗口可能比较小.

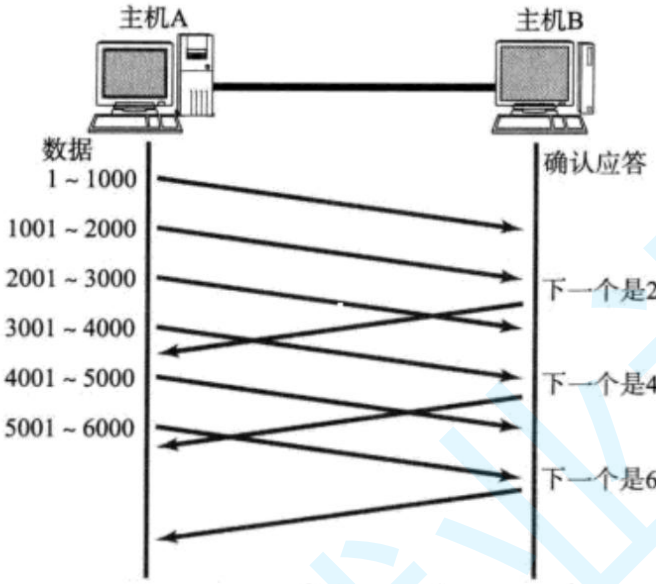
> - 假设接收端缓冲区为 1M. 一次收到了 500K 的数据; 如果立刻应答, 返回的窗口就是 500K;
> - 但实际上可能处理端处理的速度很快, 10ms 之内就把 500K 数据从缓冲区消费掉了;
> - 在这种情况下, 接收端处理还远没有达到自己的极限, 即使窗口再放大一些, 也能处理过来;
> - 如果接收端稍微等一会再应答, 比如等待 200ms 再应答, 那么这个时候返回的窗口大小就是 1M;

一定要记得, 窗口越大, 网络吞吐量就越大, 传输效率就越高. 我们的目标是在保证网络不拥塞的情况下尽量提高传输效率;

那么所有的包都可以延迟应答么? 肯定也不是;

- 数量限制: 每隔 N 个包就应答一次;
- 时间限制: 超过最大延迟时间就应答一次;

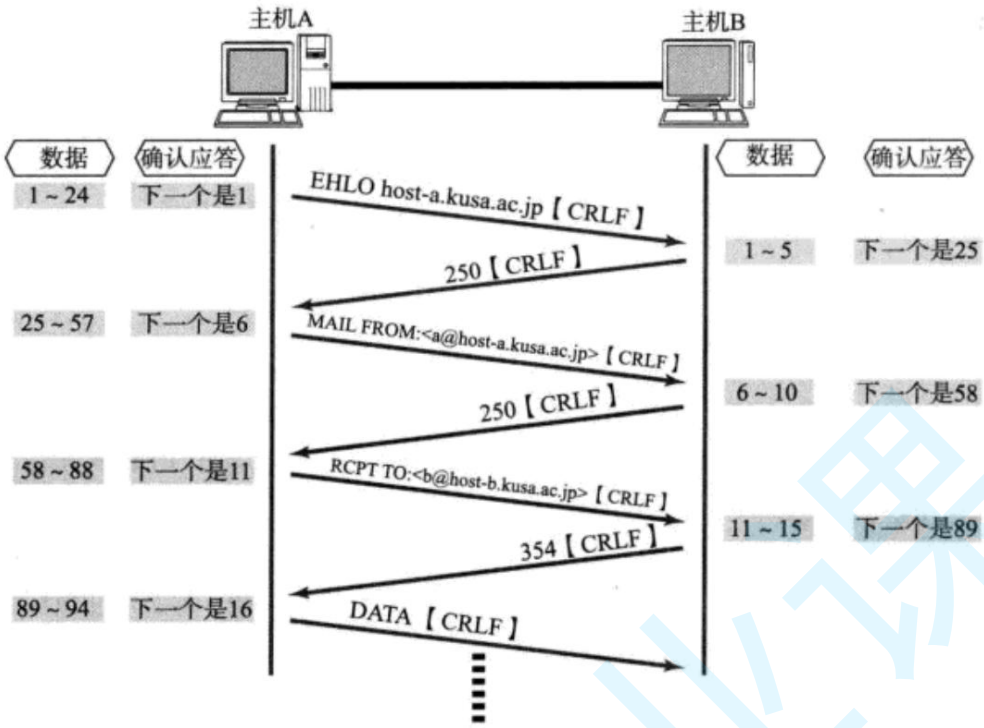
具体的数量和超时时间, 依操作系统不同也有差异; 一般 N 取 2, 超时时间取 200ms;



捎带应答

在延迟应答的基础上, 我们发现, 很多情况下, 客户端服务器在应用层也是 "一发一收" 的. 意味着客户端给服务器说了 "How are you", 服务器也会给客户端回一个 "Fine, thank you";

那么这个时候 ACK 就可以搭顺风车, 和服务器回应的 "Fine, thank you" 一起回给客户端



面向字节流

创建一个 TCP 的 socket, 同时在内核中创建一个 **发送缓冲区** 和一个 **接收缓冲区**;

- 调用 write 时, 数据会先写入发送缓冲区中;
- 如果发送的字节数太长, 会被拆分成多个 TCP 的数据包发出;
- 如果发送的字节数太短, 就会先在缓冲区里等待, 等到缓冲区长度差不多了, 或者其他合适的时机发送出去;
- 接收数据的时候, 数据也是从网卡驱动程序到达内核的接收缓冲区;
- 然后应用程序可以调用 read 从接收缓冲区拿数据;
- 另一方面, TCP 的一个连接, 既有发送缓冲区, 也有接收缓冲区, 那么对于这一个连接, 既可以读数据, 也可以写数据. 这个概念叫做 **全双工**

由于缓冲区的存在, TCP 程序的读和写不需要一一匹配, 例如:

- 写 100 个字节数据时, 可以调用一次 write 写 100 个字节, 也可以调用 100 次 write, 每次写一个字节;
- 读 100 个字节数据时, 也完全不需要考虑写的时候是怎么写的, 既可以一次 read 100 个字节, 也可以一次 read 一个字节, 重复 100 次;

粘包问题

[八戒吃馒头例子]

- 首先要明确, 粘包问题中的 "包" , 是指的应用层的数据包.
- 在 TCP 的协议头中, 没有如同 UDP 一样的 "报文长度" 这样的字段, 但是有一个序号这样的字段.
- 站在传输层的角度, TCP 是一个一个报文过来的. 按照序号排好序放在缓冲区中.
- 站在应用层的角度, 看到的只是一串连续的字节数据.
- 那么应用程序看到了这么一连串的字节数据, 就不知道从哪个部分开始到哪个部分, 是一个完整的应用层数据包.

那么如何避免粘包问题呢? 归根结底就是一句话, **明确两个包之间的边界**.

- 对于定长的包, 保证每次都按固定大小读取即可; 例如上面的 Request 结构, 是固定大小的, 那么就从缓冲区从头开始按 sizeof(Request)依次读取即可;
- 对于变长的包, 可以在包头的位置, 约定一个包总长度的字段, 从而就知道了包的结束位置;
- 对于变长的包, 还可以在包和包之间使用明确的分隔符(应用层协议, 是程序猿自己来定的, 只要保证分隔符不和正文冲突即可);

思考: 对于 UDP 协议来说, 是否也存在 "粘包问题" 呢?

- 对于 UDP, 如果还没有上层交付数据, UDP 的报文长度仍然在. 同时, UDP 是一个一个把数据交付给应用层. 就有很明确的数据边界.
- 站在应用层的站在应用层的角度, 使用 UDP 的时候, 要么收到完整的 UDP 报文, 要么不收. 不会出现"半个"的情况.

TCP 异常情况

进程终止: 进程终止会释放文件描述符, 仍然可以发送 FIN. 和正常关闭没有什么区别.

机器重启: 和进程终止的情况相同.

机器掉电/网线断开: 接收端认为连接还在, 一旦接收端有写入操作, 接收端发现连接已经不在了, 就会进行 reset. 即使没有写入操作, TCP 自己也内置了一个保活定时器, 会定期询问对方是否还在. 如果对方不在, 也会把连接释放.

另外, 应用层的某些协议, 也有一些这样的检测机制. 例如 HTTP 长连接中, 也会定期检

测对方的状态. 例如 QQ, 在 QQ 断线之后, 也会定期尝试重新连接.

TCP 小结

为什么 TCP 这么复杂? 因为要保证可靠性, 同时又尽可能的提高性能.

可靠性:

- 校验和
- 序列号(按序到达)
- 确认应答
- 超时重发
- 连接管理
- 流量控制
- 拥塞控制

提高性能:

- 滑动窗口
- 快速重传
- 延迟应答
- 捎带应答

其他:

- 定时器(超时重传定时器, 保活定时器, TIME_WAIT 定时器等)

基于 TCP 应用层协议

- HTTP
- HTTPS
- SSH
- Telnet
- FTP
- SMTP

当然, 也包括你自己写 TCP 程序时自定义的应用层协议;

TCP/UDP 对比

我们说了 TCP 是可靠连接, 那么是不是 TCP 一定就优于 UDP 呢? TCP 和 UDP 之间的优点和缺点, 不能简单, 绝对的进行比较

- TCP 用于可靠传输的情况, 应用于文件传输, 重要状态更新等场景;
- UDP 用于对高速传输和实时性要求较高的通信领域, 例如, 早期的 QQ, 视频传输等. 另外 UDP 可以用于广播;

归根结底, TCP 和 UDP 都是程序员的工具, 什么时机用, 具体怎么用, 还是要根据具体的需求场景去判定.

用 UDP 实现可靠传输(经典面试题)

参考 TCP 的可靠性机制, 在应用层实现类似的逻辑;

例如:

- 引入序列号, 保证数据顺序;
- 引入确认应答, 确保对端收到了数据;
- 引入超时重传, 如果隔一段时间没有应答, 就重发数据;
- ……

附录

```C++
// linux kernel  include/linux/tcp.h
struct tcphdr {
    __be16  source;
    __be16  dest;
    __be32  seq;
    __be32  ack_seq;
#if defined(__LITTLE_ENDIAN_BITFIELD)
    __u16   res1:4,
        doff:4,
        fin:1,
        syn:1,
        rst:1,
        psh:1,
        ack:1,
```

```
        urg:1,
        ece:1,
        cwr:1;
#elif defined(__BIG_ENDIAN_BITFIELD)
    __u16   doff:4,
        res1:4,
        cwr:1,
        ece:1,
        urg:1,
        ack:1,
        psh:1,
        rst:1,
        syn:1,
        fin:1;
#else
#error  "Adjust your <asm/byteorder.h> defines"
#endif
    __be16  window;
    __sum16 check;
    __be16  urg_ptr;
};
```

下面了解一下即可。

1. doff (Data Offset):
   - 这实际上是 TCP 头部长度（以 32 位字为单位），而不是一个标志。它指示了 TCP 头部有多少 32 位字（4 字节）。由于 TCP 头部可能包含可选项，因此这个字段用于告诉接收方 TCP 头部有多长。通常，没有可选项的 TCP 头部长度是 20 字节（即 doff 为 5，因为 5*4=20 字节）。
2. res1:
   - 这是保留位，通常被设置为 0。它们用于未来的协议扩展。
3. cwr (Congestion Window Reduced):
   - 当发送方收到一个带有 ECE（Explicit Congestion Notification，显式拥塞通知）标志的 ACK 时，它可能会设置 CWR 标志来响应。这通常用于 ECN（Explicit Congestion Notification，显式拥塞通知）机制，一种改进 TCP 拥塞控制的机制。
4. ece (ECN-Echo):
   - 当 TCP 的接收方检测到网络拥塞时，它可能会发送一个带有 ECE 标志的 ACK 给发送方。ECE 标志告诉发送方，接收方已经检测到拥塞，并且可能希

望发送方减少其发送速率。