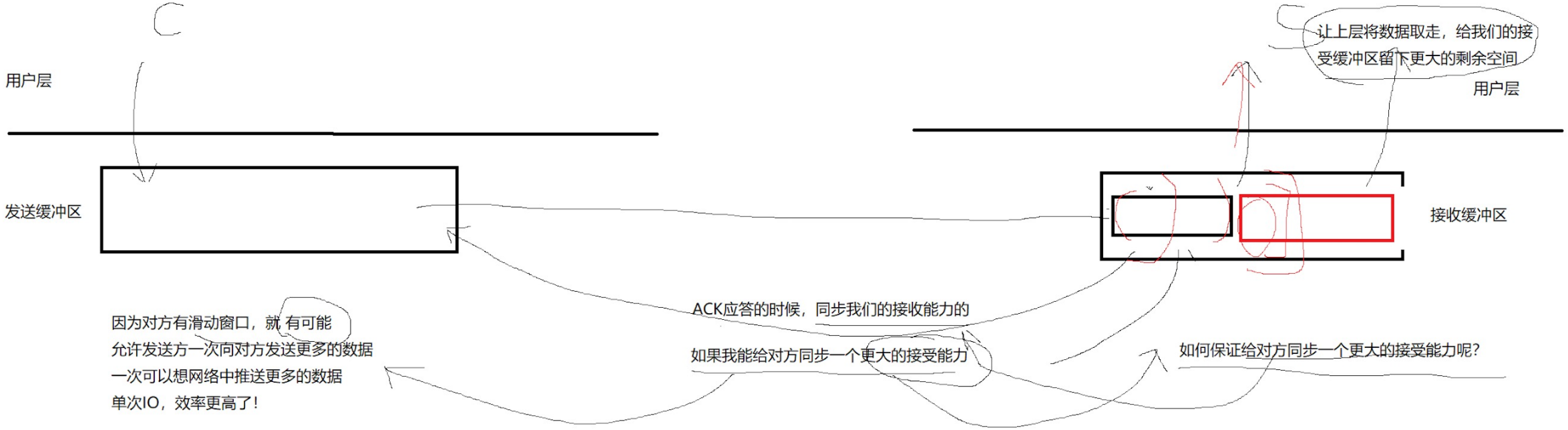
0316_Tcp



延迟应答

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;

粘包问题

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是没有区分报文和报文之间的的能力
他只向上交付了

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 相关实验

理解 listen 的第二个参数

基于刚才封装的 TcpSocket 实现以下测试代码

对于服务器, listen 的第二个参数设置为 2, 并且不调用 accept

accept要不要参与三次握手的过程呢？
不需要！
那么accept需要做什么呢？
accept是去底层获取已经建立好的连接！

如果我不调用accept，能建立连接成功吗？
可以的！
如果上层来不及调用accept，并且对端还来了大量的连接？
难道所有的连接都必须先建立好呢？

如果一开始就全部搞好，服务器肯定挂的

现在我们把listen的第二个参数设置为1

```
18  #include "Log.hpp"
19
20  class Sock
21  {
22  private:
23      const static int gbacklog = 1; // 一般不能太大也不能太小，后面再详细解释
24  public:
25      Sock() {}
26      ~Sock() {}
27      int Socket()
28      {
29          int listen_sock = socket(AF_INET, SOCK_STREAM, 0);
30          if (listen_sock < 0)
```

```
8   int main()
9   {
10      Sock sock;
11      int listen_sock = sock.Socket();
12      sock.Bind(listen_sock, 8080);
13      sock.Listen(listen_sock);
14      while(true)
15      {
16          std::cout << "Listen Statue... " << std::endl;
17          sleep(1);
18      }
19      return 0;
20  }
```

现在我的上层也只调用Listen
不调用accept
所以只让服务器一直处于listen的状态

```
(base) [yufc@ALiCentos7:~/Src/BitCodeField/0316]$ ./TcpServer
[NORMAL] [1684210978] create socket success, sock: 3
[NORMAL] [1684210978] init TcpServer Success
Listen Statue...
Listen Statue...
Listen Statue...
Listen Statue...
```

```
 will not be shown, you would have to be root to see it all.)
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      -
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN      -
tcp        0      0 0.0.0.0:8080            0.0.0.0:*               LISTEN      2035/./TcpServer
tcp        0      0 127.0.0.1:34835         0.0.0.0:*               LISTEN      1067/node
tcp        0      0 127.0.0.1:34835         127.0.0.1:43908         ESTABLISHED 1067/node
```

处于listen状态
这个也没问题

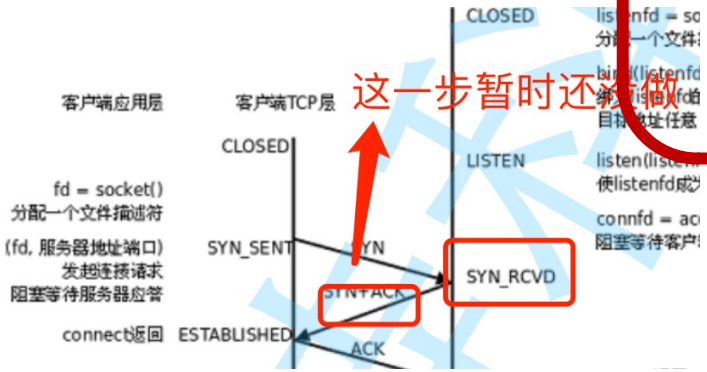
调试控制台 终端 端口

终端

```
(base) [yufc@VM-12-12-centos:~/Core]$ telnet 47.120.41.234 8080
Trying 47.120.41.234...
Connected to 47.120.41.234.
Escape character is '^]'.
```

然后！我们去连第三个的时候
发现状态不再是ESTABLISHED了
而是SYN_RECEIVE状态

表明三次握手没有完成！
SYN_RECEIVE的那个，他会等一会儿
如果等一会还没轮到他连，那么他就会直接走人，直接退出



我们用另一台机子telnet连上
继续连，连多几个！
我们发现连两个的时候
两个服务都是ESTABLISHED状态
表明都连上了（注意，此时listen的第二个参数是1）

因此，listen的第二个参数的意义：
就是
底层全连接队列的长度 = listen的第二个参数+1

这是因为, Linux内核协议栈为一个tcp连接管理使用两个队列:

1. 半链接队列（用来保存处于SYN_SENT和SYN_RECV状态的请求）
2. 全连接队列（accpetd队列）（用来保存处于established状态，但是应用层没有调用accept取走的请求）

而全连接队列的长度会受到 listen 第二个参数的影响.

全连接队列满了的时候, 就无法继续让当前连接的状态进入 established 状态了.

这个队列的长度通过上述实验可知, 是 listen 的第二个参数 + 1.