选择重发 ARQ 算法模拟

组别: 第十组
成员: 19030100036 董津玮
19180100042 成　诺

1　程序设计思路

1.1　选择重发 ARQ 算法

在重发 ARQ 下，**发送方**（下文称：sender）和**接收方**（下文称：receiver）各自具有一套工作算法，双方通过这对算法在**有噪声信道**（下文称：channel）中完成通信。

在教材中可以找到两种算法的伪程序实现，但是这份伪代码在工程视角上具有一些逻辑错误。比如判断接收到的数据帧是否在窗口内的边界条件不明晰、接收到数据帧后清楚缓存的相关逻辑位置不太准确以及接收到帧后判断帧的准确性和判断 NAK 发送标志的逻辑不能放在一个 IF 判断中……

1.2　程序实现思路

发送方和接收方的工作流程大体与伪代码一致，但是在实现中增加了信道（channel）这个模型，因为部分传输错误（如比特翻转）以及传播时延发生在物理信道中，个人认为如果需要全面而细致的模拟这个模型需要将信道单独建模。至此模型的大体框架已经确定，即：**Sender-Channel-Receiver 模型**。这个模型下的三个子模型正好对应了三个模块（sender.py、channel.py 和 receiver.py），三个模块间通过本地的 socket 建立 TCP 连接模拟 bit 流的传递。由于 TCP 是面向字节的，所以在成帧时所使用的协议也是面向字节的协议

为了可视化的展示 Sender-Channel-Receiver 模型，需要把每个对象抽象出一些**动作（action）**，用这些动作来反映出整个通信的工作流程（注意这些动作不是传输算法中的“Event“）：

Sender 具有以下动作：成帧、发送数据帧、重传（超时重传，NAK 重传）、数据帧丢失；

Receiver 具有以下动作：向上传递数据、帧损坏、发送 NAK 帧、发送 ACK 帧、NAK 帧丢失、ACK 帧丢失；

Channel 具有以下动作：帧传递、比特翻转、超时。

所有的这些动作以时间戳为优先值，通过 HTTP 请求保存到服务端的优先队列中，前端轮询优先队列，将这些事件渲染成相应的图形显示在浏览器界面上，完成可视化过程。

为了能和用户进行交互，设计了一套指令系统，通过 HTTP 请求下达指令后，使得程序在执行过程中检查到指令为真时可以触发响应动作。设计的三个指令分别是：

“损坏（damage）”：控制 Channel 在传递下一个帧时进行字节翻转，损坏帧中某些数据；

“丢失（loss）”：控制 Sender 或 Receiver 模拟发送后丢失某些帧的情况；

“超时（timeout）”：控制 Channel 突然增加下一次 Forward 数据时的延时，触发超时重传机制。

1.3 程序实现细节

1.3.1 成帧

之前说了，成帧使用的是面向字节的协议。该协议油头部、有效数据载荷以及尾部组成。头部是一个字节的类型字段和一个字节的 seq 字段，有效数据载荷是由网络层取得的数据，尾部是对前面所有字段的 CRC32 校验。在发送时还会在帧的首尾添加一个 FLAG 字节用于区分帧的起始和终止为了区分帧中正常数据和 FLAG，在成帧发送时，会在每个 ESC 字节和 FLAG 字节前插入一个转义字节 ESC，而在接收方取出帧时则会去掉这些多余的转义字节。

```
frame_fmt_header = "!BB"
frame_fmt_tail = "!I"

# 单个帧最大携带MAX_FRAME_DATA字节数据，传入多余的会被截断
frame = struct.pack(
    frame_fmt_header, FRAME_TYP_DATA, self.Sn % self.seq_range)  # 头部字段
frame += data  # Payload
u32crc = zlib.crc32(frame) & 0xffffffff
frame += struct.pack(frame_fmt_tail, u32crc)  # 尾部crc32
print("[Sender] Make a frame from data:", data)
self.__report_make_frame_action(self.Sn % self.seq_range, data.hex(), u32crc)
return frame
```

```
def __media_send_frame(self, frame: bytes):
    # 插入转义字符
    def pre_process(data: bytes):
        new_data = b""
        for byte in data:
            if byte == ESC or byte == FLAG:
                new_data += struct.pack("!BB", ESC, byte)
            else:
                new_data += struct.pack("!B", byte)
        return new_data
    # 尝试在信道上发送帧
    frame = pre_process(frame)
    data = struct.pack("!B", FLAG) + frame + struct.pack("!B", FLAG)  # 加上起止标志字节
    try:
        self.channel.send(data)
    except Exception as e:
        print("[Sender] Channel is not accessible:", e)
```

1.3.2 发送方算法

（详细见附件 sender.py 中的 Sender.__main_loop 方法）

首先在发送方设置一个 threading.Event 对象和 threading.Timer 对象，然后开启一个物理层监听线程和网络层监听线程。这两个线

程代表着不同的事件（如：有数据需要发送以及帧到达）。在启动后__main_loop 首先进入 Event.wait()状态等待事件唤醒。当线程中的事件发生时，线程首先获取事件锁，并写入事件类型到变量中，然后触发 threading.Event——此时__main_loop 会被唤醒进入激活状态，向下执行并判断发生了什么事件，从而进入相应的逻辑。在处理完事件后，__main_loop 会释放事件锁并重新进入 Event.wait()状态等待下一个事件发生

1.3.3 信道算法

这里的逻辑比较简单，channel 模块充当了一个透明代理的作用，帮助双方交换数据。首先监听两个端口分别用于与 Sender 和 Receiver 的通信。待两个端口都有主机上线后进入 daemon 循环中，通过 select 方法取出 Sender 和 Receiver 两个 socket 中的可操作对象，并通过读和写双方 socket 两个步骤来交换数据。

```
while True:
    s_rlist, s_wlist, _ =  select.select(rlist, wlist, xlist)

    # sender forward to receiver
    if self.sender in s_rlist and self.receiver in s_wlist:
        buf = self.sender.recv(MAX_BUF_SIZE)
        if len(buf)>0:
            buf = self.__damage(buf)
            print("[Channel] Forward data to receiver:", buf)
            self.__send_with_timeout("receiver", buf)
    # receiver forward to sender
    if self.receiver in s_rlist and self.sender in s_wlist:
        buf = self.receiver.recv(MAX_BUF_SIZE)
        if len(buf)>0:
            buf = self.__damage(buf)
            print("[Channel] Forward data to sender:", buf)
            self.__send_with_timeout("sender", buf)
```

1.3.4 接收方算法

（详细见附件 receiver.py 中的 Receiver.__main_loop 方法）

总体工作流程与发送方一致，但是事件的类型不同，这里只有一种事件就是帧到达事件。

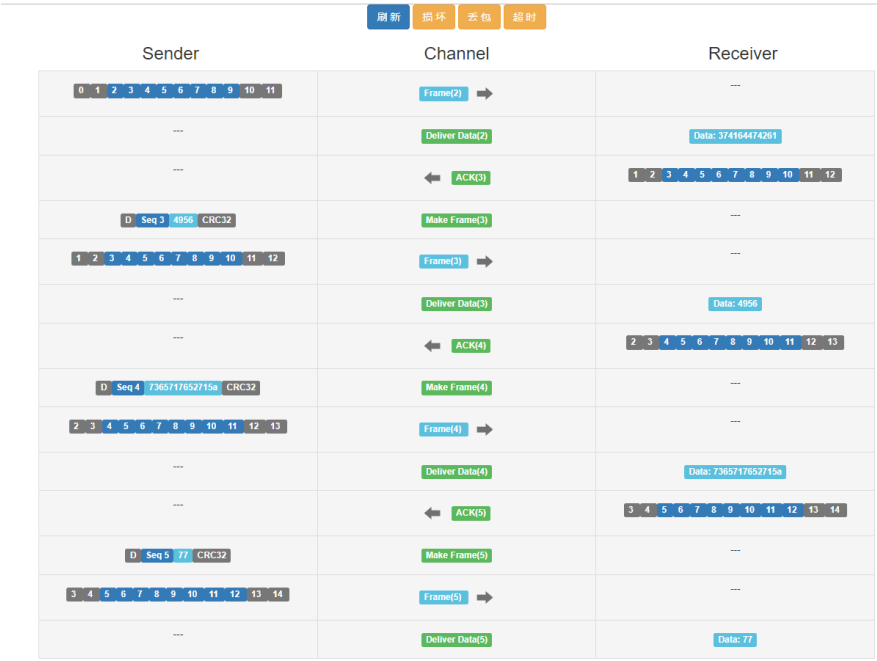
1.3.5 窗口维护算法

发送方和接收方都有各自的窗口，具有相应的维护操作。发送方主要是控制 Sf 和 Sn 两个指针，而接收方则主要控制 Rn 指针。虽然教材说理论上这三个指针的值都是模序列最大长度的。但是在一些范围判断的代码中，如果一直保持这几个值模序列最大长度会非常麻烦，于是引入了 seq 变量，通过模序列最大长度得到 seq 的值。范围判断使用 Sf、Sn 和 Rn，成帧等的操作则使用 seq 变量。

2 运行效果演示

2.1 界面介绍

2.1.1 整体界面



2.1.2 顶部按钮

顶部四个按钮用于交互控制，刷新按钮用于刷新页面数据。另外三个按钮对应上文提到的三个人工指令，点击指令按钮即可设置服务端上的全局变量，触发一次帧损坏、帧丢失或超时重传。注意触发一次效果后该全局变量会被自动置 0，避免引发长时间混乱。

2.1.3 滚动列表

滚动列表分为三个大列，左边的列显示 Sender 的内部信息，如：发送窗口和成帧信息等；中间的列表示 Channel 上发生的事件，如：帧发送、帧丢失、帧错误以及成帧等信息；右边的列显示 Receiver 的内部信息如：接收窗口模型以及从帧中取出并传递给上层的数据；

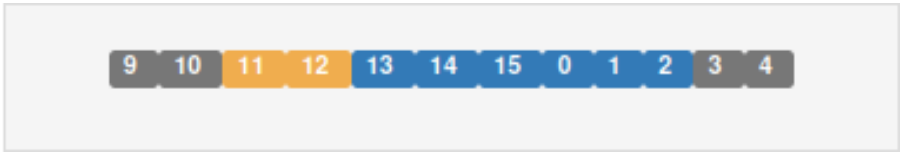
滚动列表的滚动方向是向上滚动，最新的帧会出现在最下面。

2.1.4 成帧

发送方生成一个新的帧会在滚动列表中按照字段（除了发送时添加的 FLAG 标识）显示，其中比较关键的字段是帧序号和帧携带的上层数据。该数据可以和接收方最后传递给网络层的数据做对比判断数据是否被正确传达。

2.1.5 窗口

发送方窗口：



灰色部分为非窗口区域，黄色和蓝色为窗口区域，黄色表示**已发送未确认**，蓝色表示**可发送帧**

接收方窗口：

灰色部分为非窗口区域，绿色和蓝色为窗口区域，蓝色表示**窗口缓冲区**，绿色表示**乱序到达**的帧

2.2 实例

2.2.1 正常数据帧发送

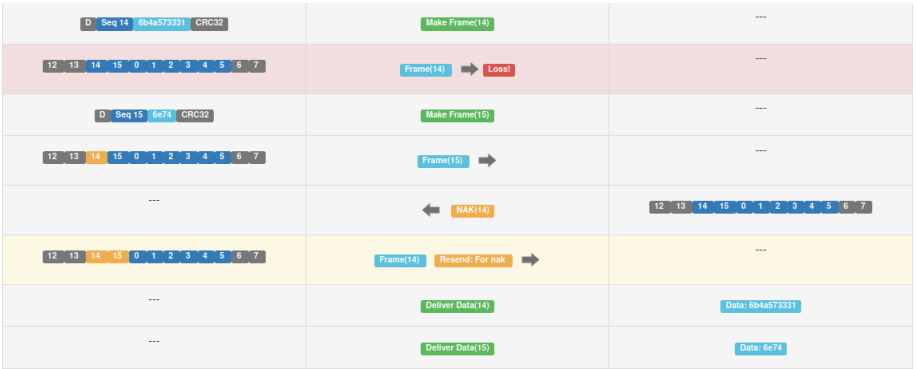
截取一个正常的帧发送样例：

成帧 - 发送帧 - 接收帧（向上传递） - ACK

可以看到

2.2.2 数据帧丢失

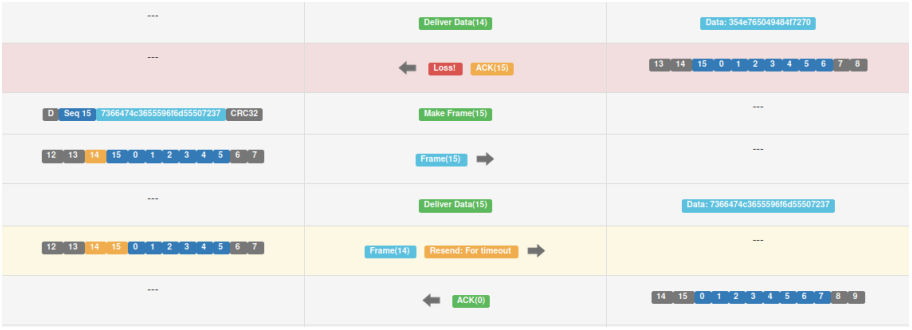
下面是手动模拟一个数据帧在传输时丢失的样例：



可以看到样例中，seq 为 14 的帧被丢失，发送方没有发现而继续发送 seq 为 15 的帧。由于乱序到达，接收方发送了一个 NAK 提醒发

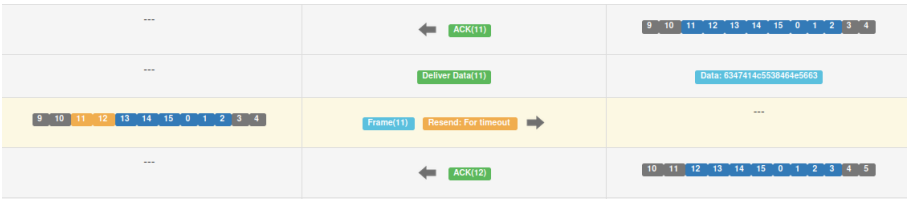
送方，发送方重发了 seq 为 14 的帧，每个重发的帧会标明重发原因。此图中重发原因是“For Nak”，和之前的过程相呼应。重发后 seq 为 14 的帧的数据被正常接收并传递。

2.2.3 ACK 帧丢失



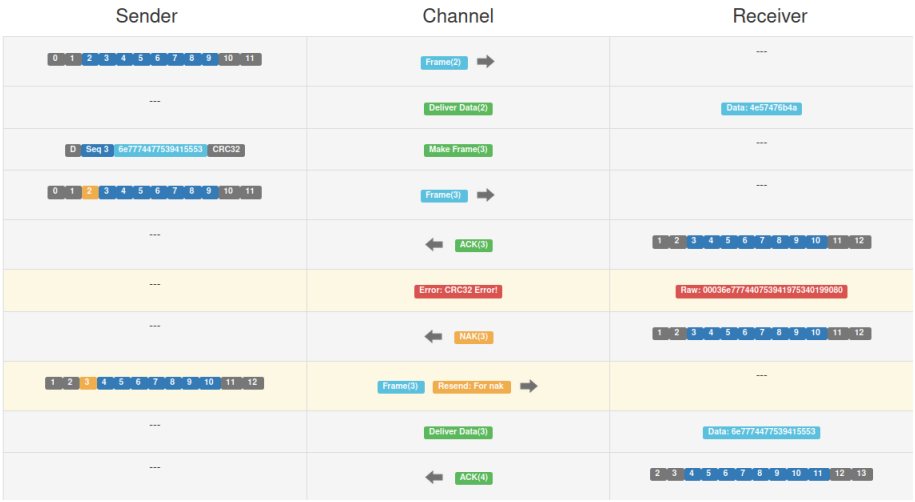
可以看到 ACK(15)丢失后，由于一直没有确认，发送方重发了 seq 为 14 的帧。

2.2.4 ACK 超时



可以看到在 ACK(12)超时未到达时，发送方收到超时信号便重发了 seq 为 11 的帧。

2.2.5 数据帧被损坏



可以看到，接收方接收到数据帧后会主动去校验 CRC32 的值，发现值不正确后直接将该帧丢弃，此时的效果和数据帧丢失是一样的，直到下次发送方重发且未丢失和错误后该帧才被正确传递。