> 第一次写这个作业的时候遇到了很大的困难，思路不清楚，也没有很好的切入点。做了很久越做越乱，于是干脆重做了……
>
> 第二次动手的时候整理出一下思路。

# 1 实验要求

1）包括实验过程和实验结果。

2）要包含实验和所学知识点的对应关系，实验对所学知识点的体现和运用，比如实验中如何实现定时器、如何实现重传，如何进行差错检查等。

3）对于Go-Back-N，如果N的值设为1，则相当于停等式（Stop-Wait）传输，比较不同N值对传输速率的影响。

4）请设置不同的丢包率、误比特率、带宽、传播时延，模拟不同类型的通信信道，分析不同信道情况下，如何设置报文大小和窗口大小N，实现传输速率最大化。

5）实验过程中的碰到的问题和解决过程，实验的心得、体会和收获。

# 2 示例代码的错误更正

在`corruptPacket`函数的代码中，显然可以看到编译器提示表达式恒为0。这个表达式显然是没有意义的。

由整个项目的内容可以知道，packet[0], packet[1]用来生成 seq num，packet[2]是数据的第一个 byte，packet[packet.length-1]是最后一个byte (但不一定属于数据的部分，因为数据的长度是随机生成指定的，而 packet.length 是一个固定大小的值)。

个人猜测是破坏了数据部分的第一个byte，最后一位是checksum，用来做检验。

`Math.random()`返回的是一个0~1之间的浮点数，转化为byte类型默认为0。这里正确的表达式应该是`(byte)Math.(Math.random() * 256)`，这样生成的就是一个0~256的浮点数，转化为byte刚好是一个八位的二进制随机数。

```java
private void corruptPacket( byte[] packet){  //packetSize 单位：byte
    double packetErrorRatio = 1 - Math.pow(1 - bitErrorRatio, (packet.length * 8));

    //corrupt packet
    if (Math.random() < packetErrorRatio) {
        if (packet[1] == 0) {
            packet[0] = (byte)(-packet[0]);
        } else {
            packet[1] = (byte) (-packet[1]);
        }
        packet[2] = (byte)Math.random();
        packet[packet.length - 1] = (byte)Math.random();
    }
}

//计算端到端延迟 = 发送延迟 + 传播延迟，单位：毫秒，忽略了排队和处
private long calcE2EDelay(int packetSize) {
```

Result of '(byte)Math.random()' is always '0'

'Math.random()' cast to 'byte' is always rounded down to '0'

Replace with '0' Alt+Shift+Enter  More actions... Alt+Enter

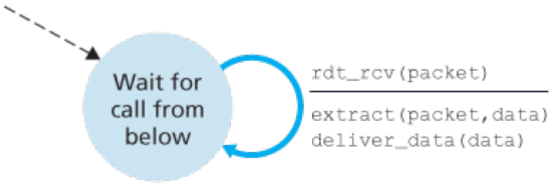
实现过程中存在思路不清楚的情况，这里决定一步一步完成。

# 3 rdt 1.0的实现

rdt 1.0 假设信道是可靠的，相比 udt 没有增加什么新功能。其 FSM 图如下，整体内容是很简单的。

udt的部分在示例代码中已经完成了，不需要实现。由于可见，关于sending side，我们需要完成 make_pkt 方法，然后在 rdt_send 方法中调用。关于 receiving side，需要完成 extract 和 deliver_data 方法，然后在 rdt_rcv 中调用。


a. rdt1.0: sending side




b. rdt1.0: receiving side

我的实践过程如下：

- OpenChannel。需要设置对方的ip，并打开udtchannel。
- closeChannel。需要调用udtchannel，并且关闭。
- make_pkt，考虑到之后至少还有checksum在最后一位，因此限制生成报文的长度，这里设置为了 1000 和 50。
- rdt_send。先直接调用udt的功能，之后再完善。
- rdt_rcv。得到了 length 和 buffer 后，就相当于已经完成了 extract 的工作（已经知道前几位是数据，故相当于提取出来了）。同时因为这里是java中的引用，应用层和传输层都指向同一个对象，因此相当于传递到了应用层。extract 和 deliver_data 也就隐式地实现了。

经过测试，代码可以正常运行。

# 4 UdtChannel 的完善

- 示例代码的 UdtChannel 是单向的。经过确认，示例代码中，仅仅给出了12345一个接收端口。由于是在一台 host 上验证GBN的实现，由于 socket 是由一个四元组构成的，因此 sender 和 receiver 不能是在同一个端口接收报文。因此，需要对发送方和接收方做不同的处理。

需要做的改进如下：

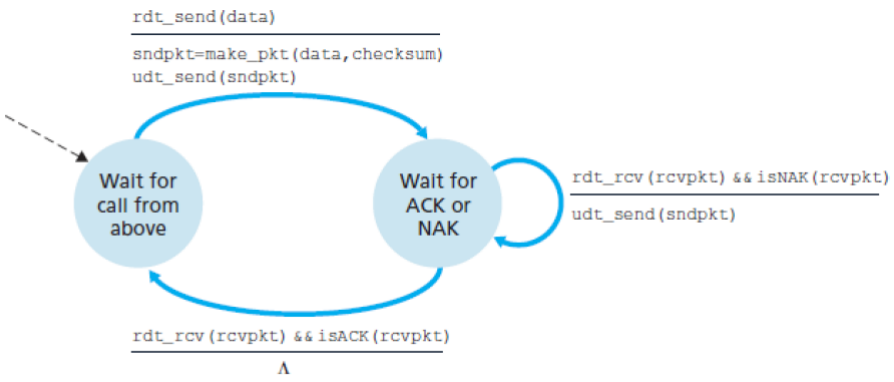
1. 改写UdtChannel的构造方法，对于发送方和接收方，分别将其接收端口设置为 12345 和 12346。
2. 改写UdtSend方法，对于发送方和接收方，分别将其 peerPort 设置为 12346 和 12345。

经过检验，代码可以正常运行。

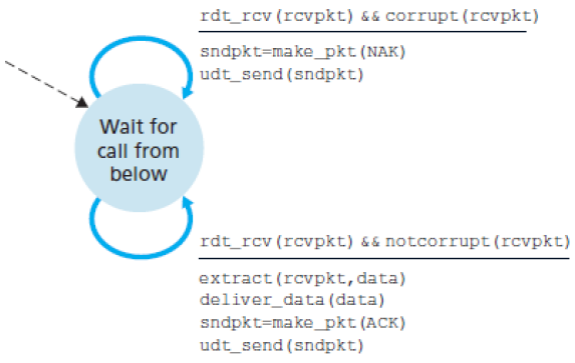
至此，我们建立了一个可以双向通信的UdtChannel，同时可以通过 rdt 1.0 来调用它。接下来就可以实现 GBN 特有的功能了。

# 5 GBN的实现

GBN的FSM图如下。实现过程主要参考此图。




a. rdt2.0: sending side




b. rdt2.0: receiving side

相比 rdt 1.0，需要增加如下功能：

## 5.1 错误检测

Error detection，错误检测。即需要实现 checksum 检查错误。对于发送端，应当在 `make_pkt` 方法中，计算checksum。在接收端，应当可以计算实际checksum，并与理论checksum进行比较。因此可以写一个 `IsCorrupted` 方法来判断。

**在实验过程中发现如下问题**：在示例代码中，发送一个报文前，生成一个和长度一样大小的`byte[]`，但是这样，就只能存储数据部分，没有位置存储checksum了。为了方便实现，统一设置长度为1440的`byte[]`。数据部分最多占1000位，checksum放在最后一位(即`byte[1439]`)，长度放在倒数第二、第三位。如图，注意`data.length != length`。关键代码如下：

```
public void make_pkt(byte[] data, int length){
    long checksum = 0;
    for(int i = 2; i < length; i++)
        checksum += data[i];
    data[data.length-1] = (byte) checksum;
    // 数据的长度，用两位byte存储。
    data[data.length-3] = (byte) (length % 128);
    data[data.length-2] = (byte) (length / 128);
}
```

接收端需要首先从最后三位读取checksum和长度信息，再去计算和比较。关键代码如下：

```
public boolean isCorrupted(byte[] packet){
    int length = packet[packet.length - 2] * 128 + packet[packet.length - 3];
    long checksum = 0;
    for(int i = 2; i < length; i++)
        checksum += packet[i];
    if((byte)checksum == packet[packet.length - 1])
        return false;
    else return true;
}
```

# 5.2 结果反馈

反馈结果。

1. 首先，需要发送确认。在rdt_rcv里实现。如果报文没出错，且恰好是接收方想要收的报文，则expectedNum++。同时需要通过udtSend发送确认。
2. 如果没有出错，则再发一遍确认。
3. 发送端需要根据接受到的ACK更新 base。该过程在rdtRecvAckAction中实现。根据FSM图实现即可。

```
int len = buffer[buffer.length - 2] * 128 + buffer[buffer.length - 3];
int seq = buffer[1] * 128 + buffer[0];
if (!isCorrupted(buffer) && seq == expectedNum){
    expectedNum++;
    byte[] packet = new byte[3];
    packet[0] = buffer[0];
    packet[1] = buffer[1];
    packet[2] = -1;
    udtChannel.udtSend(packet, length: 3);
}else {
    byte[] packet = new byte[3];
    packet[0] = (byte) (expectedNum % 128);
    packet[1] = (byte) (expectedNum / 128);
    packet[2] = -1;
    udtChannel.udtSend(packet, length: 3);
    rdtSend(packet, len: 3);
```

**在实验过程中发现如下问题**：

在debug的过程中，发现expectedNum符合预期，但是发送端一直在发送同一个报文，排查发现是因为在报文出错/非等待报文的情况下，没有选择发确认。这是因为没有理解GBN，没有考虑到发送的确认也存在丢失的情况。

# 5.3 数据重传

### 5.3.1 定时器的实现

数据重传。需要实现定时器。使用了TimerTask，到了时间后就会执行run。

```
// Timer 是一个定时器，启动定时器，规定时间后就会执行run
class Timeout extends TimerTask {
    Timer timer;
    public Timeout(Timer timer){
        this.timer = timer;
    }

    @Override
    public void run() {
        sender_schedule(RdtEvent.RDT_EVENT_TIMEOUT, new byte[MAX_RDT_PKT_SIZE], MAX_RDT_PKT_SIZE);
    }
}
```

start_timer 相当于一个Timeout的构造方法。

stop_timer 只需删除这个对象即可。需要注意的是，由于timer是一次性的，所以删除后需要再new一个。

### 5.3.2 重传方法的实现

在这个方法的实现上遇到了两个问题：

1. 线程阻塞很严重，而且有exception。后来检查发现，是因为rdtTimeoutAction和rdtSendAction可能会同时进行，两个方法都会占用udtChannel，同时操作的情况下，可能会有异常。因此设置了flag：IsTimeOutInProcess。在TimeoutAction中发送报文时，通过这个flag禁止rdtSendAction发送，这样就能避免同时占用一个channel。
2. flag：IsTimeOutInProcess设置的位置不合适。为了避免重复无效地重传，在重传之后，使该线程等待一段时间再继续。但是我把flag设置在等待结束以后再恢复正常，那么在这段等待时间里，rdtSendAction本该可以执行，但也不能执行。这样就导致传输速率非常慢，极大地影响了传输速度。

   修正以后，传输速度显著提高了。

   可以说是差之毫厘，谬以千里。看似执行顺序无关紧要的两行代码对执行效率的影响非常大。

```
    try {
        int next = nextSeqNum;
        int b = base;
        // 执行timeout的flag，由于 rdtsend 也要使用udtChannel，
        // 为了防止二者同时占用，因此需要设置flag：IsTimeOutInProcess
        IsTimeOutInProcess = true;
        // 发送任务结束后不需再执行重传
        for (int i = b; i < next; i++) {
            byte[] packet = new byte[MAX_RDT_PKT_SIZE];
            packet[0] = (byte) (i % 128);
            packet[1] = (byte) (i / 128);
            int datasize = genDataBlockSize();
            make_pkt(packet, datasize);
            // 下面这句用于测试，与GBN无关
            packet[packet.length - 4] = (byte) 233;
            udtChannel.udtSend(packet, datasize);
            if (i == b)
                start_timer();

            IsTimeOutInProcess = false;
            System.out.println("Retransmit packet: " + b + " to " + (nextSeqNum-1));
        }
        Thread.sleep( millis: N * 25);
```

# 6 其他问题

## 6.1 提前结束的Channel

前文所述的问题解决后，还是会出现问题：

当报文发送到最后的时候 (例如，总共要发100个packet，N=5，当发送到96个报文，进入最后一组报文的时候)，会一直重传同一组报文。设置测试语句后，发现最后重传的这一组报文，一个都没有收到，更没有返回ACK，因此一直在重传。经过检查发现，这是因为发送方的在第一次发送完100个报文后，没有等待全部确认，就结束了主函数，直接执行了rdtProtocol.closeChannel，这样双方的连接就断了，重传的线程发送的报文，就无法被接收了。

为了解决这个问题，我在主程序外增加了一段代码，用来等待直到全部确认结束。关键代码如下。

```
while (rdtProto.base != packet_num){
    // 全部发送后，也不一定全部收到，可能还要经过重传才能接收到。
    // 在这里，必须等待，否则就会执行后面的closechannel，一旦关闭了通道，
    // 后面的重传就不可能收到了。
}
```

## 6.2 命名以及代码规范问题

1. rdtProtocol中的endType类型与udt中的定义不一致，因此做了统一。
2. 诸如LOCAL_IP，senderPort等常量，事先定义好了，使代码一定程度上做到了自注释。

# 7 结果分析

1. **对于Go-Back-N，如果N的值设为1，则相当于停等式（Stop-Wait）传输，比较不同N值对传输速率的影响。**

本题中各参数的设置如图所示。

```
public RdtProto(UdtChannel.UdtEndType endType) {
    commEndType = endType;
    udtChannel = new UdtChannel(endType);
    udtChannel.setBandwidth(1); //设置带宽为1Mbps
    udtChannel.setLossRatio(3); //设置丢包率为3%
    udtChannel.setBRT(0.5);      //设置误比特率为万分之0.5
    udtChannel.setPropDelay( lower: 5, upper: 15);
}
```

统一发送100个报文，其时间如图所示：

| 行数 | 时间（毫秒） |
| --- | --- |
| 1 | 5629 |
| 3 | 6272 |
| 5 | 10628 |
| 7 | 13267 |
| 9 | 17655 |

由以上十几组对照实验的结果可以看出，停等协议的传输效率是最高的。这是因为，程序模拟了丢包和误比特率的情况。在数据的传输过程中会出现错误。出错重传的时间代价可以如此思考：

1. 对于停等协议，如果出现了错误，会马上检测出来，并让发送方及时把这个报文补发过来。
2. 对于GBN协议，如果出现了错误，nextSeqnum只要不超过窗口大小，依然会++，也就是依然会发送新的报文，但是这些报文是无效的，会被refuse。这些报文的发送是一种对网络资源的浪费，因此会降低速率。

由此可见，出错重传的时间代价，GBN协议远比停等协议要大。

2. **请设置不同的丢包率、误比特率、带宽、传播时延，模拟不同类型的通信信道，分析不同信道情况下，如何设置报文大小和窗口大小N，实现传输速率最大化。**

若网络质量比较不错，各项参数均有一定提升：

| 丢包率 / 100 | 误比特率 /10000 | 带宽 Mbps | 传播时延 | N | 时间 (毫秒) |
|---|---|---|---|---|---|
| 1 | 0.1 | 5 | 1-5 | 1 | 4399 |
| 1 | 0.1 | 5 | 1-5 | 3 | 1495 |
| 1 | 0.1 | 5 | 1-5 | 5 | 1913 |
| 1 | 0.1 | 5 | 1-5 | 7 | 3401 |
| 1 | 0.1 | 5 | 1-5 | 9 | 4105 |

不难看出，时间有了明显变化。其中，停等协议变成了耗时最长的。N=3时耗时最短，往后随着N增大逐步提高。

- 当网络质量比较不错的时候。出错重传的次数比较少，大多数时候只需发一次即可。在这一情况下，停等协议必须等待一个ACK发过来，才能发下一个，等待的时间代价就会很大，发送方的利用率很低。因此耗时就会更长。
- 当网络质量比较差的时候（例如上一题的条件），经过检查，发现出错的报文非常多，这种时候，就可以使用停等协议，尽量避免重传，从而提高发送方利用率。

# 8 实验体会

这个实践作业难度比我一开始预期要大很多。

在第一次尝试解决的时候，我比较靠直觉，根据自己对GBN的理解实现，没有太多章法，再加上缺乏多线程的代码编写经验，在实践过程中碰到了很大困难。由于第一次缺乏章法，又把原本的项目改得面目全非，预计自己引入了不少错误，遂放弃。

第二次尝试解决的时候，我首先把老师的示例代码理清楚了。发现示例代码虽然是正确运行的，但是建立的是单向的连接，不能直接运用到rdt中。同时我进一步参阅了课本上关于GBN的说明，确定了实现的思路。按照思路逐步实行：首先建立起发送方和接收方的连接，然后实现rdt调用udt，最后实现GBN特有的功能。加入了很多测试语句来验证代码的正确性，最终才完成了实验。

在实践过程中，我深刻体会到编写代码是一件非常考验逻辑性、非常严谨的工作。由于本次实验是在模拟现实状况，因此代码编写的状况要尽可能贴合实际。两个语句的顺序不同，所模拟的场景是完全不同的。

**收获**

1. 先想清楚再写代码，不然很浪费时间。
2. 代码逻辑要严谨，不是差不多就可以。
3. 对JAVA编程有一定提升。
4. 对GBN有了更深的理解。