# 目录

# 1 实验内容说明

## 1.1 实验题目

实验 3：基于 UDP 服务设计可靠传输协议并编程实现

## 1.2 实验说明

实验 3-2：在实验 3-1 的基础上，将停等机制改成基于滑动窗口的流量控制机制，采用固定窗口大小，支持累积确认，完成给定测试文件的传输。

# 2 协议设计

使用滑动窗口进行流量控制，实现选择重传。

## 2.1 数据包格式

为了实现可靠传输，首先设计数据包格式，不仅包括停等协议，也为后续实验做准备。


图 1: 数据包格式

标志位 FLAG 分别设置了：ACK，SYN，FIN，SEQ 四项，window 为接收方缓冲区大小。协议中，每个序列号对应一个字节。

## 2.2 建立连接

连接建立过程如下：

1. 发送方向接收方发送一个数据包 A(SEQ=0,FLAG=FIN)，告知起始序列号。
2. 接收方回复数据包 B(ACK=A.SEQ,FLAG=ACK,Window=32KB)，告知接受缓存区大小。
3. 发送方接受数据包 B 后成功连接。

此外，发送方接收超时会重传一定次数。

## 2.3 发送与确认

发送与确认过程如下：

1. 发送方发送一个数据包 A(SEQ=x,FLAG=SEQ,data)，告知序列号，数据，数据长度。
2. 接收方回复按序接受的最后一个序列号，并缓存失序数据，即回复数据 B(ACK=last.SEQ,FLAG=ACK,window=y)。
3. 发送方根据 B 中 ACK，调整发送序列号。

此外，发送方会对每个数据包设置定时器，接受超时时会进行重传。

## 2.4 关闭连接

关闭连接过程如下：

1. 发送方等待发送缓存区空后，发送数据包 A(SEQ=x,FLAG=FIN)，告知序列号，通知接受方关闭。
2. 接收方回复按序接受的最后一个序列号，数据包 B(ACK=last.SEQ,FLAG=ACK,window=y)。若 FIN 未失序，则关闭。
3. 发送方接受到 x 被确认则关闭。

此外，发送方会对每个数据包设置定时器，接受超时时会进行重传。

# 3 滑动窗口实现

## 3.1 发送端

### 3.1.1 概述

发送端采用事件循环模型，发送端发送数据过程中的事件及行为如表1所示。

| 事件 | 动作 |
|---|---|
| 应用层提交数据 | 检查发送缓存区，未满时缓存数据。同时，检查接收方窗口，窗口未满时发送 |
| 接收到 ACK | 更新序列号，弹出缓冲区中已被确认的数据同时，检查接收方窗口，窗口未满时发送 |
| 数据包超时 | 重新发送数据包 |

表 1: 发送端事件响应

### 3.1.2 定时器实现

使用单线程实现，实现过程利用按时间排序的优先队列。工作线程在接受数据包前，会先处理时间事件队列。期间，对超时的事件进行数据包的重传，并返回到下一个计时器触发的事件间隔。工作线程处理完时间事件队列，后非阻塞接受数据包，超时时间为到下一个定时器发送。若非阻塞等待期间接受到数据包，则处理数据包，否则处理超时的计时器。

</> CODE 1: 定时器数据结构

```
//时间事件数据结构
struct Timevt {
    chrono::time_point<chrono::system_clock> tp;//触发时间
    uint32_t dataSeq;  //序列号
    uint16_t dataLen;  //从序列号开始的，数据长度
    uint8_t nTimeout;  //已经超时的次数
    uint8_t flag;      //数据包标记:SEQ/SYN/FIN
};

//事件队列
priority_queue<Timevt,vector<Timevt>,greater<Timevt>> timevts;

//添加计时器
void RSend::setTimer(uint32_t seq, uint16_t size,uint8_t nTimeout,
   uint8_t flags)
{
    auto timeout = milliseconds(getRTO().count()<< nTimeout);
    timevts.emplace(seq, size, timeout, nTimeout, flags);
}
```

</> CODE 2: 定时器实现

```
milliseconds RSend::handleTimer()
{
    if (timevts.empty()) return milliseconds(10);
    auto now = system_clock::now();
    while (now > timevts.top().tp) {  //计算是否触发计时器
        processTimeout(timevts.top());
        timevts.pop();
        if (timevts.empty()) return milliseconds(10);
    }
    //返回现在到下一个时间事件的时间间隔
    return duration_cast<milliseconds>(timevts.top().tp - now);
}
```

### 3.1.3 流量控制实现

当应用层提交数据后，或当有新数据被确认后，工作线程会检查发送缓存区与发送窗口，并据此选择是否发送数据。当前还可以发送的数据大小，取一下三者的最小值:

- 接受缓冲区数据空余数据长度，即 sendbase+window-nextseq
- 未发送已缓存的数据长度，即，buffer.end() - nextSeq
- 最大段长度，即 MSS

</> CODE 3: 滑动窗口的流量控制实现

```
buffer.begin();  //发送缓存区起始序列号，即sendbase
buffer.end();    //发送缓存区结尾序列号，即已缓存的最大序列号+1。
window ;         //接受缓冲区大小，根据接收端的回复进行更新
nextSeq;         //已发送的最大序列号+1。
void RSend::trySendPkg()
{
    while (true) {
        //先确定可发送的数据大小
        int sendSize = min(
            min(buffer.begin() + window - nextSeq/*发送窗口限制*/,
                buffer.end() - nextSeq/*不超过发送缓冲区已有数据*/),
            MSS                        /*不超过最大段长度*/
        );
        if (!(sendSize>0)) break;
        sendSize=sendSeq(nextSeq, sendSize); //发送数据包
        setTimer(nextSeq, sendSize,0,RPkg::F_SEQ);
        nextSeq += sendSize;        //发送后更新已发送的序列号
    }
}
```

## 3.2 接收端

### 3.2.1 概述

接收采用事件循环模型，接收端发送数据过程中的事件及行为如表2所示。

| 事件 | 动作 |
|---|---|
| 接收数据包 | 数据包有序则合并缓存、确认数据；数据包失序则缓存数据、确认有序数据。 |
| 应用层取回数据 | 缓冲区弹出数据，更新窗口。 |

表 2: 接收端事件

### 3.2.2 更新窗口

窗口的值即接收缓存区的空余尺寸的值。接收方会在回复的确认告知发送方当前窗口大小。

</> CODE 4: 接收端窗口更新

```
uint16_t getWindow() { return buffer.freeSize(); }
```

### 3.2.3 累积确认

当接收到完整的失序数据后，需要合并已缓存数据。同时对有序的数据包进行累积确认。

</> CODE 5: 累积确认

```
//seq       失序数据包起始字节的序列号
//size      失序数据包字节长度
//nextACK   已确认的最大序列号

//首先判断该有序数据包是否所有数据都已经被确认过
uint32_t end = seq + size;
if (end <= nextACK)
    return;
//获得该数据包未被确认的数据起始地址的偏移，以及需要写入的数据长度
uint32_t dataOffset = nextACK - seq;
uint32_t wSize = end - nextACK;
//已缓存的失序数据避免重复写入
if (!cacheRanges.empty()) {
    uint32_t nextCachedBegin = cacheRanges.front().first;
    wSize = min(wSize, nextCachedBegin - nextACK);
}
//写入缓存区，保证数据不溢出，返回实际写入长度
uint32_t pushnum=buffer.push(data+dataOffset,wSize);
nextACK += pushnum; //根据写入长度，更新最后已确认确认的序列号
```

### 3.2.4 缓存失序数据

对于失序的数据包需要进行缓存，当接收到所有数据包后需要对缓存数据进行合并和确认。

对于失序的数据包，在缓存区有空余的情况下，将其写入到缓冲区对应位置，并记录所有已经被缓存的位置。对于已缓存的数据，新的失序数据包会覆盖写入。

</> CODE 6: 缓存失序数据

```
//seq       失序数据包起始字节的序列号
//size      失序数据包字节长度
//data      失序数据包数据地址
//nextACK   已确认的最大序列号

//获得seq在buffer中的对应位置。
//当发送端起始seq为0时，buffer.end()==nextACK
uint32_t begin = buffer.end() + seq - nextACK;
//写入缓存区，返回实际写入长度。写入时保证缓冲区不溢出。
uint32_t setnum = buffer.set(begin,data,size);
auto range = make_pair(seq, seq + setnum);      //已缓存的范围

//按range.first排序插入已缓存范围列表
auto iter = cacheRanges.begin();
while (iter!= cacheRanges.end()
       && iter->first < range.first)
    iter++;
cacheRanges.insert(iter,range);
```

对于这些被缓存的数据，需要在接收到完整失序数据后进行合并。只要保证所有的缓存范围有序记录，就可以方便的进行合并。

</> CODE 7: 合并失序数据

```
//cacheRanges 保存所有已缓存的序列范围，按范围起点排序
//nextACK   已确认的最大序列号

while (!cacheRanges.empty()) {  //合并缓存
    uint32_t nextCachedBegin = cacheRanges.front().first;
    uint32_t nextCachedEnd = cacheRanges.front().second;
    if (nextACK < nextCachedBegin) break; //if 数据不完整不能合并

    if (nextACK < nextCachedEnd) { //if 该范围未被覆盖
        //合并缓存，更新接收缓冲区
        uint32_t pushnum = buffer.push(nextCachedEnd - nextACK);
        nextACK = nextCachedEnd;  //更新已确认序列号
    }
    cacheRanges.pop_front();
}
```

# 4 效果演示

## 4.1 文件发送

发送端在命令行参数中输入接受方 IP,UDP 端口, 发送文件名称。发送后显示统计

```
.\Sender.exe -i 127.0.0.1 -p 12300 -f 2.jpg
...
send file spent 0.713325 s.
5898511 bytes in total. speed = 8269037 bytes ps
success to send the file: 2.jpg.
```

接收端同样在命令行参数中指明 IP 和 UDP 端口。最终成功发送文件。

```
.\Recipient.exe -i 127.0.0.1 -p 12300
...
succeed to close connect.
recv file successfully, file:C:/Users/A/Desktop/net/recvfile/2.jpg
```

## 4.2 日志记录

发送端日志详细记录了 buffer 变化，超时重传，接受数据包，当前各项参数等状态信息。

```
...
[LOG]: RECV ACK ack=5849094, nextSeq=5877766, window=32768,
buffer[5844998,5877766)
[LOG]: POP 4096 bytes from buffer. buffer:[5849094,5877766).
[LOG]: <USER> PUSH 4096 bytes into buffer.
[LOG]: GET 4096 bytes from buffer .
[LOG]: SEND SEQ [5877766,5881862).
[LOG]: TIMER [5877766,5881862) in 10 ms, 0 times,flag=16.
[LOG]: STATE nextSeq=5881862,sendBase=5849094,window=32768,
toSend=0, buffer:[5849094,5881862).
...
```

接收端同样有详细日志。

```
...
[LOG]: recv-->recv | window=32768, nextACK=5869574,
buffer=[5869574,5869574).
[LOG]: RECV [5869574,5873670), flags=16.
[LOG]: PUSH 4096 bytes into buffer, buffer:[5869574,5873670).
[LOG]: <USER> recv 4096 bytes.
[LOG]: SEND ACK ,ack=5873670, window=32768.
...
```