# SR协议 (Selective Repeat, 选择重传协议)

## 协议简介

SR协议是一种滑动窗口协议，相比GBN协议更高效。主要特点：

### SR vs GBN 对比

| 特性 | GBN协议 | SR协议 |
| --- | --- | --- |
| 接收方缓冲 | 无缓冲，只接受按序到达的包 | 有缓冲，可接收乱序包 |
| 重传策略 | 超时重传整个窗口 | 只重传丢失的包 |
| ACK机制 | 累积确认 | 独立确认每个包 |
| 效率 | 丢包时效率低 | 丢包时效率高 |
| 复杂度 | 实现简单 | 实现复杂 |

## 主要改进

### 1. 发送方改进

* **独立计时器**：每个包有独立的超时计时
* **选择性重传**：只重传超时的包，不影响其他已确认的包
* **窗口管理**：使用map存储每个包的状态（是否确认、发送时间、重传次数）

### 2. 接收方改进

* **接收缓冲区**：使用map缓存乱序到达的数据包
* **接收窗口**：维护一个接收窗口，接受窗口内的任何包
* **按序交付**：缓存的数据按序交付给应用层
* **独立ACK**：对每个正确接收的包都发送ACK

### 3. 关键数据结构

**发送方窗口状态：**

struct SendWindow {  
 bool acked; // 是否已确认  
 clock\_t sendTime; // 发送时间(用于超时判断)  
 int retryCount; // 重传次数  
};  
map<int, SendWindow> sendWindow; // 序列号 -> 状态

**接收方缓冲区：**

struct RecvBuffer {  
 DataFrame frame; // 数据帧  
 bool received; // 是否已接收  
};  
map<int, RecvBuffer> recvWindow; // 序列号 -> 缓冲

## 使用方法

### 编译

# 编译服务器  
g++ -g server.cpp -o server.exe -lws2\_32 -fexec-charset=GBK -finput-charset=UTF-8  
  
# 编译客户端  
g++ -g client.cpp -o client.exe -lws2\_32 -fexec-charset=GBK -finput-charset=UTF-8

### 运行

1. **启动服务器**

* ./server.exe

1. **启动客户端**

* ./client.exe

1. **测试SR协议**

* -testsr [数据包丢失率] [ACK丢失率]
* 示例：
* -testsr 0.2 0.2 # 20%数据包丢失率，20% ACK丢失率  
  -testsr 0.3 0.1 # 30%数据包丢失率，10% ACK丢失率  
  -testsr 0 0 # 无丢包测试

## 工作流程

### 发送方流程

1. 初始化发送窗口 [sendBase, sendBase+windowSize)  
2. 发送窗口内的所有未发送包  
3. 循环：  
 - 检查是否有包超时，超时则重传该包  
 - 接收ACK：  
 \* 标记对应包为已确认  
 \* 如果是sendBase，滑动窗口  
 - 发送新的包（如果窗口允许）  
4. 所有包确认完成，传输结束

### 接收方流程

1. 初始化接收窗口 [recvBase, recvBase+windowSize)  
2. 循环接收数据：  
 - 收到包seq：  
 \* 在窗口内：缓存并发送ACK  
 \* 在窗口前：发送ACK（重复包）  
 \* 在窗口后：丢弃  
 - 尝试交付：  
 \* 从recvBase开始，按序交付所有连续的包  
 \* 更新recvBase，滑动窗口  
3. 收到最后一个包且已交付，传输结束

## 关键参数

#define SEND\_WIND\_SIZE 10 // 发送窗口大小  
#define RECV\_WIND\_SIZE 10 // 接收窗口大小  
#define SEQ\_SIZE 20 // 序列号空间 (必须 >= 2\*窗口大小)  
#define TIMEOUT\_THRESHOLD 100 // 超时阈值(ms)  
#define MAX\_RETRIES 10 // 最大重传次数

**重要**：序列号空间必须至少是窗口大小的2倍，以避免序列号混淆问题。

## 优势场景

SR协议在以下场景下比GBN更优：

1. **高丢包率**：只重传丢失的包，不浪费带宽
2. **大窗口**：窗口越大，SR的优势越明显
3. **长延迟**：重传单个包的代价小于重传整个窗口

## 示例输出

### 服务器端：

[发送] Seq=0 (包0), 数据="数据包1:SR协议测试开始"  
[发送] Seq=1 (包1), 数据="数据包2:Selective-Repeat测试"  
...  
[ACK处理] 收到ACK=0  
[确认] Seq=0 已确认  
[窗口滑动] 新的sendBase=1  
...  
[超时重传] Seq=5 (第1次)

### 客户端端：

[接收] Seq=0, 数据="数据包1:SR协议测试开始"  
 [缓存] 存入接收缓冲区  
 [发送] ACK=0  
 [交付] Seq=0, 数据="数据包1:SR协议测试开始"  
 [窗口滑动] 新的recvBase=1  
...  
[接收] Seq=7, 数据="数据包8:提高传输效率"  
 [缓存] 存入接收缓冲区  
 [发送] ACK=7  
 [接收窗口] recvBase=5, 范围: [5, 14]  
 [×××√××××××] # Seq=7已接收，但recvBase=5还未收到

## 注意事项

1. **序列号回绕**：使用模运算处理序列号循环
2. **缓冲区管理**：及时清理已交付的数据，避免内存泄漏
3. **超时设置**：根据网络延迟调整TIMEOUT\_THRESHOLD
4. **窗口大小**：确保 SEQ\_SIZE >= 2 \* WINDOW\_SIZE

## 文件结构

SR\_protocol/  
├── server.cpp # SR协议服务器端实现  
├── client.cpp # SR协议客户端实现  
└── README.md # 本说明文档

## 实验建议

1. **对比测试**：在相同丢包率下测试GBN和SR的性能差异
2. **调整参数**：尝试不同的窗口大小和超时阈值
3. **极端情况**：测试高丢包率(50%+)下的表现
4. **网络延迟**：增加Sleep时间模拟高延迟网络

## 技术细节

### 序列号空间问题

SR协议要求序列号空间 ≥ 2×窗口大小，原因：

* 发送窗口最多占用N个序列号
* 接收窗口最多占用N个序列号
* 需要能区分新旧数据包

### 超时机制

每个包独立计时，使用clock()函数：

double elapsed = (double)(now - sendTime) \* 1000.0 / CLOCKS\_PER\_SEC;  
if (elapsed >= TIMEOUT\_THRESHOLD) {  
 // 重传  
}

### 窗口滑动条件

* **发送窗口**：当sendBase对应的包被确认时滑动
* **接收窗口**：当recvBase对应的包被接收并交付后滑动