

南开大学

计算机网络实验报告

Lab3-3



学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

摘要

本次实验专注于选择重传 (Selective Repeat, SR) 协议的实现, 其中一个关键的关注点是发送和接收窗口大小的不匹配问题。实验要求中提到的“发送窗口和接收窗口采用相同大小”的要求在具体实践上是有问题的。即在某些极端情况下, 接收方的窗口可能完全领先于发送方的窗口。当所有发送方窗口内的数据包都被接收方接收, 但相应的确认 (ACK) 还未被发送方收到时。在这种情况下, 发送窗口和接收窗口采用相同大小会导致程序陷入死循环。针对这一问题, 实验进行了调整, 将接收端窗口大小设定为发送端窗口大小的两倍。这种改变基于对 SR 协议效率和可靠性的深入考虑, 特别是在处理重传和失序数据包时。实验中对此进行了必要的代码实现, 特别强调了接收端设计部分, 以确保高效和准确的数据传输。这一改进有助于提高整体的网络性能, 特别是在高误码率或网络延迟大的环境中。详细的代码实现和逻辑可以在附带的压缩包中的 'server.cpp' 文件中找到。通过这种优化实现, 实验不仅满足了基本的 SR 协议要求, 而且在保持协议的高效性和可靠性方面迈出了重要一步。

(注意: 调整的部分在于蔡学长讨论以后, 我认识到这部分的优化并非绝对必要。在选择重传协议中, 接收端针对每个分组发送的确认 (ACK) 是基于各自的序列号的。这意味着, 即使接收端的窗口已经滑动并且之前接收到的分组确认信息丢失, 接收端依旧能够根据相应的序列号向发送端回复相应的 ACK。因此, 对窗口大小的这种优化可能并不是必需的。感谢蔡学长提供的指导, 这有助于我更准确地理解 SR 协议的工作机制。)

关键字: Selective Repeat; 协议设计; 流水线机制

目录

一、 实验要求	1
二、 选择重传协议 (Selective Repeat) 原理介绍	1
(一) SR 协议发送端行为	1
(二) SR 协议接收端行为	1
(三) SR 与 GBN 的区别	2
三、 选择重传协议 (Selective Repeat) 协议设计	2
(一) 报文格式	2
(二) 发送端设计	2
1. 发送分组独立确认	2
2. 失序到达 ACK 的处理	3
3. 动态窗口管理	4
4. 超时重传	4
(三) 接收端设计	5
1. 序列号处理	5
2. 接收确认 (ACK) 的发送	5
3. 窗口管理	6
4. 数据包的缓存与交付	7
(四) 丢包测试设计	7
1. 发送分组丢包	8
2. 接收确认 (ACK) 丢包	8
(五) 其他部分	9
(六) 补充说明: 窗口大小与序列号范围的关系	9
四、 协议优化部分	9
(一) 问题指出	9
(二) 优化实现	10
五、 优点与不足	10
(一) 优点	10
(二) 不足	10
六、 结果展示	11
(一) 实验结果	11
(二) 三次握手	11
(三) 丢包测试	12
(四) 四次挥手	12
七、 实验心得与体会	12

一、 实验要求

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，发送窗口和接收窗口采用相同大小，支持选择确认，完成给定测试文件的传输。

- 协议设计：数据包格式，发送端和接收端交互，详细完整
- 流水线协议：多个序列号
- 发送缓冲区、接收缓冲区
- 选择确认：SR(Selective Repeat)
- 日志输出：收到/发送数据包的序号、ACK、校验和等，发送端和接收端的窗口大小等情况，传输时间与吞吐率
- 测试文件：必须使用助教发的测试文件（1.jpg、2.jpg、3.jpg、helloworld.txt）

二、 选择重传协议 (Selective Repeat) 原理介绍

选择重传 (Selective Repeat, SR) 协议是一种用于确保可靠数据传输的高效机制，特别适用于误码率高或延迟变化大的网络环境。SR 协议通过允许发送端和接收端独立处理每个数据包，优化了数据传输的效率和可靠性。

(一) SR 协议发送端行为

- **数据发送和窗口管理**：发送端维护一个固定大小的滑动窗口。当有新数据需要发送时，如果下一个序列号在窗口内，数据被封装成分组并发送。发送的每个分组在发送前被缓存，以备可能的重传。
- **定时器和重传**：每个发送的分组都有一个关联的定时器。如果在定时器超时前未接收到相应的确认 (ACK)，则该分组将被重传。
- **ACK 处理**：收到 ACK 后，标记相应分组为已确认，并停止其定时器。如果 ACK 是窗口左边缘的 ACK，窗口向前移动，允许发送更多数据。如果 ACK 不是窗口左边缘的 ACK，窗口则不移动。

(二) SR 协议接收端行为

- **数据包接收和确认**：接收端的滑动窗口接收序号在特定范围内的数据包。对于每个正确接收的数据包，发送端发送相应的 ACK。
- **数据缓存和交付**：接收端缓存失序到达的数据包，直到可以按序交付给上层应用。当收到窗口左边界的数据包时，窗口向前移动，并交付所有连续的数据包。
- **重复数据包的确认**：对于已经确认和交付的重复数据包，接收端仍会发送 ACK，防止因发送端未收到 ACK 而触发不必要的重传。

(三) SR 与 GBN 的区别

- **窗口和确认机制:** SR 协议允许每个数据包独立确认, 而 GBN 协议只对最高序列号的连续数据包发送单一确认。
- **重传策略:** SR 协议只重传未被确认的分组, 而 GBN 在超时后重传整个窗口内所有未被确认的分组。
- **失序数据包处理:** SR 允许接收端接收并缓存失序的数据包, 而 GBN 会丢弃任何失序的数据包。

三、 选择重传协议 (Selective Repeat) 协议设计

(一) 报文格式

```
1 typedef struct Packet_Header
2 {
3     BYTE WindowSize;
4     BYTE seq_quotient;
5     ...
6 }
```

- 在 TCP 协议的头部中, 第 24 到 31 位被标记为窗口 (WindowSize) 字段。该字段用于定义滑动窗口的大小。在协议的实现中, 发送方利用这个窗口字段来指示当前滑动窗口的可用空间大小。
- 在 TCP 协议的头部中, 第 40 到 47 位被标记为“序列号商” (seq_quotient) 字段。由于 seq 大小为 8 字节, 可支持的序列号范围为 0 到 255, 当序列号大小超过 255 时, 则会进入循环, 因此, 接收方收到的 seq 是分组序列号模 256 以后的结果。为了在接收端复原该分组在发送端真正的序列号, 加入“序列号商”字段, 即接收端分组真正的序列号为 $seq + 256 \times seq_quotient$
- 其余部分详见实验 3-1

(二) 发送端设计

1. 发送分组独立确认

在 SR 协议中, 每个发送的数据包都有一个唯一的序列号, 并且每个数据包的确认 (ACK) 是独立的。这意味着每个数据包可以独立地被确认, 不依赖于其他数据包的确认状态。发送方会根据收到的每个 ACK 独立更新其状态。以下代码片段展示了如何处理接收到的 ACK, 以及如何基于独立的 ACK 更新发送窗口:

```
1 memcpy(&packet, receiveBuffer, sizeof(packet));
2 if ((compute_sum((WORD*)&packet, sizeof(packet)) != 0)) {
3     continue;
4 }
5 int packetIndex = int(packet.ack) % WindowSize;
6 if (!ackReceived[packetIndex]) {
```

```

7      ackReceived[packetIndex] = true;
8      timerRunning[packetIndex] = false;
9      cout << "\033[32m[ACC]\033[0m 收到确认 \033[0mACK:\033[0m" << int(packet.ack) << ",
      对应窗口索引:\033[0m" << packetIndex << endl;
10     while (ackReceived[base % WindowSize] && base < nextSeqNum) {
11         ackReceived[base % WindowSize] = false;
12         timerRunning[base % WindowSize] = false;
13         base++;
14     }
15 }
16 else {
17     cout << "\033[33m[WARNING]\033[0m 收到重复确认 \033[0mACK:\033[0m" << int(packet.ack)
      << endl;
18 }

```

在这段代码中，当收到一个新的 ACK 时，发送方首先验证其有效性。如果有效，发送方更新对应序号数据包的确认状态。此外，发送方还会检查是否可以滑动其发送窗口，这是通过确认窗口基序号 base 之后连续的数据包是否已被确认来决定的。这种独立的确认机制允许发送方更加高效地利用网络资源，因为它可以在不必等待整个窗口内的所有数据包都被确认的情况下，发送新的数据包。独立确认的优势在于其提高了协议的吞吐量，并减少了由于网络延迟和数据包丢失导致的重传。这一特性尤其在网络条件复杂或变化大的环境中显得尤为重要。

2. 失序到达 ACK 的处理

在 SR 协议中，发送方接收到的 ACK 可能不是按顺序到达的。这意味着某些后续的数据包可能先于之前的数据包被确认。发送方需要能够处理这种情况，确保每个数据包独立地被确认，并根据 ACK 适当地调整其窗口。以下代码片段展示了如何处理接收到的 ACK，以及如何基于这些 ACK 滑动窗口：

```

1  int packetIndex = int(packet.ack) % WindowSize;
2  if (!ackReceived[packetIndex]) {
3      ackReceived[packetIndex] = true;
4      timerRunning[packetIndex] = false;
5      cout << "\033[32m[ACC]\033[0m 收到确认 \033[0mACK:\033[0m" << int(packet.ack) << ",
      对应窗口索引:\033[0m" << packetIndex << endl;
6      while (ackReceived[base % WindowSize] && base < nextSeqNum) {
7          ackReceived[base % WindowSize] = false;
8          timerRunning[base % WindowSize] = false;
9          base++;
10     }
11 }
12 else {
13     cout << "\033[33m[WARNING]\033[0m 收到重复确认 \033[0mACK:\033[0m" << int(packet.ack)
      << endl;
14 }

```

在这个片段中，首先通过校验和检查接收到的 ACK 的有效性。然后，根据 ACK 的序号更新对应数据包的确认状态。如果接收到的 ACK 确认了窗口中最早的未确认数据包，发送方将滑动窗口。这是通过检查 ackReceived 数组来完成的，该数组跟踪了每个数据包的确认状态。即使某些

数据包的 ACK 先于其他数据包到达 (即 ACK 失序), 该方法也能确保窗口只在连续的数据包被确认后才向前移动。这允许发送方有效地处理在网络中可能发生的数据包乱序情况。

3. 动态窗口管理

动态窗口管理主要体现在根据接收到的 ACK 来调整窗口大小和位置。发送方维护一个固定大小的窗口, 该窗口内的分组可以被发送。当分组被确认后, 窗口会向前滑动, 允许发送更多的数据。

```

1  if (!ackReceived[packetIndex]) {
2      ackReceived[packetIndex] = true;
3      timerRunning[packetIndex] = false;
4      while (ackReceived[base % WindowSize] && base < nextSeqNum) {
5          ackReceived[base % WindowSize] = false;
6          timerRunning[base % WindowSize] = false;
7          base++;
8          cout << "\033[35m[MOVING]\033[0m 收到顺序到来的确认号-窗口滑动-从
              << base << "到" << base + WindowSize << endl;
9      }
10 }

```

在这个片段中, 当接收到新的 ACK 时, 发送方首先检查该 ACK 是否是重复的。如果不是重复的 ACK, 则根据该 ACK 的序号更新窗口的状态。接下来, 发送方检查 ackReceived 数组, 以确定是否可以滑动窗口。如果窗口的最左边的分组已被确认 (即 ackReceived[base % WindowSize] 为 true), 则窗口向前滑动。这允许发送方发送新的数据包, 从而利用网络带宽。动态窗口管理机制的优势在于它能够适应网络条件的变化。例如, 在网络状况良好时, 确认信息能够快速到达发送方, 使得窗口可以频繁地向前滑动, 从而加快数据的发送速度。反之, 在网络状况较差时, 窗口的滑动会减慢, 以避免过多的数据包在网络中积压。动态窗口管理在 SR 协议中起着至关重要的作用, 它使得发送方能够根据接收方的处理能力和网络状况灵活地调整数据发送策略。

4. 超时重传

在 SR 协议中, 每个发送的数据包都有一个与之关联的定时器。如果在定时器超时之前未收到对应的 ACK, 数据包将被重传。这个机制确保了即使数据包在网络中丢失或延迟, 也能够被有效地重新发送。以下代码片段展示了超时重传机制的具体实现:

```

1  for (int i = base; i < nextSeqNum; i++) {
2      if (!ackReceived[i % WindowSize] && (clock() - packetTimers[i %
        WindowSize]) / CLOCKS_PER_SEC > RESEND_INTERVAL) {
3          sendto(socketClient, stagingBuffer[i % WindowSize], sizeof(packet) +
              stagingBufferLengths[i % WindowSize], 0, (sockaddr*)&servAddr,
              servAddrlen);
4          cout << "\033[31m[WARNING]\033[0m 没有接收到ack, 触发超时-重传分
              组, 未确认包编号: " << i << endl;
5          packetTimers[i % WindowSize] = clock();
6      }
7  }

```

在这个片段中，发送方定期检查每个已发送但尚未收到 ACK 的数据包。如果某个数据包的定时器超时（即当前时间与 `packetTimers[i % WindowSize]` 的差值超过了预设的重传间隔 `RE-SEND_INTERVAL`），则该数据包将被重新发送。发送后，相应的定时器被重置以便再次跟踪其超时状态。超时重传机制对于在不可靠的网络环境中保持数据传输的可靠性至关重要。在网络条件不理想、数据包丢失率高的情况下，这种机制能够确保所有数据包最终都能被成功地传输到接收方。

（三） 接收端设计

选择重传 (Selective Repeat, SR) 协议的接收端设计关键在于正确处理接收到的数据包，发送 ACK 确认，并适当地管理接收窗口。以下是关于接收端关键特性的详细介绍：

1. 序列号处理

在 SR 协议中，接收端必须能够正确地识别每个数据包的序列号，即使在面对高序列号和可能的溢出时也不例外。正确处理序列号是确保数据包能够被正确地接收和确认的关键。

```
1 if (packet.tag == 0 && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {  
2     int seqNum = int(packet.seq) + int(packet.seq_quotient) * 256;  
3     int bufferIndex = seqNum % (2 * WindowSizeSize);  
4 }
```

在这段代码中，接收端首先通过校验和检查接收到的数据包的有效性。然后，它计算数据包的实际序列号。由于序列号可能会溢出（例如，如果使用了较小的数据类型来存储序列号），因此代码使用了两部分来表示序列号：`packet.seq` 和 `packet.seq_quotient`。这两部分结合起来提供了一个更大范围的序列号空间，允许接收端正确处理较大的序列号。

2. 接收确认 (ACK) 的发送

在 SR 协议中，接收端需要为每个接收到的数据包发送 ACK，无论这些数据包是否按顺序到达。这确保了发送方可以知道哪些数据包已经被接收，并据此更新其发送窗口。以下代码片段展示了如何在接收到数据包后发送 ACK：

```
1 if (packet.tag == 0 && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {  
2     int seqNum = int(packet.seq) + int(packet.seq_quotient) * 256;  
3  
4     bool isInDoubleWindowSize = (seqNum >= rcv_base - WindowSizeSize) && (  
5         seqNum < rcv_base + WindowSizeSize);  
6  
7     if (isInDoubleWindowSize) {  
8         Packet_Header ackPacket;  
9         ackPacket.tag = 0;  
10        ackPacket.ack = seqNum;  
11        ackPacket.checksum = 0;  
12        ackPacket.checksum = compute_sum((WORD*)&ackPacket, sizeof(ackPacket));  
13        memcpy(receiveBuffer, &ackPacket, sizeof(ackPacket));  
14        sendto(socketServer, receiveBuffer, sizeof(ackPacket), 0, (sockaddr*)&clieAddr, clieAddrLen);  
15    }
```



```

14     cout << "\033[33m[SEND]\033[0m发送确认信息-确认号:" << int(
        ackPacket.ack) << ",校验和:" << int(ackPacket.checksum) << endl
        ;
15 }
16 // 其他逻辑...
17 }

```

在这个片段中，接收端首先对接收到的数据包进行校验。如果数据包有效，接收端会检查该数据包的序列号是否在其可接收的窗口范围内（即 2 倍窗口大小范围内）。对于有效的数据包，接收端创建一个 ACK 数据包，其中包含了接收到的数据包的序列号。然后，接收端发送这个 ACK 数据包回发送方。发送 ACK 的这一机制对于 SR 协议至关重要，因为它允许发送方独立地确认每个数据包的接收状态。这一机制确保了即使数据包失序到达，发送方也能得知每个数据包的接收状态，并据此管理其发送窗口。ACK 发送在 SR 协议的接收端中扮演着核心角色，它允许接收端对每个接收到的数据包发送独立的确认，从而提高了整个通信过程的可靠性和效率。

3. 窗口管理

在 SR 协议中，接收端维护一个固定大小的窗口，用于控制可以接收的数据包的序列号范围。当收到在窗口内的数据包时，接收端将其缓存，并根据需要调整窗口的位置。

```

1  if ((seqNum >= rcv_base) && (seqNum < rcv_base + WindowSizeSize)) {
2      if (!isReceived[bufferIndex]) {
3          memcpy(packetBuffer[bufferIndex], receiveBuffer + sizeof(packet),
4                  packet.datasize);
5          isReceived[bufferIndex] = true;
6          cout << "\033[36m[INFO]\033[0m数据包成功接收并缓存-序列号:" <<
7              seqNum << endl;
8
9          if (seqNum == rcv_base) {
10             while (isReceived[rcv_base % (2 * WindowSizeSize)]) {
11                 memcpy(Message + totalDataLength, packetBuffer[rcv_base % (2
12                     * WindowSizeSize)], packet.datasize);
13                 totalDataLength += packet.datasize;
14                 isReceived[rcv_base % (2 * WindowSizeSize)] = false;
15                 rcv_base++;
16                 cout << "\033[35m[MOVING]\033[0m收到顺序到来的确认号-窗口
17                     滑动-从" << rcv_base << "到" << rcv_base +
18                     WindowSizeSize << endl;
19             }
20         }
21     }
22 }
23 // 处理重复数据包的逻辑...
24 }

```

在这段代码中，接收端首先检查每个接收到的数据包是否落在当前接收窗口内。如果数据包在窗口内，并且之前未被接收（即未被标记为已接收），则接收端将其内容复制到缓冲区，并标记该序列号为已接收。关键的部分是对窗口左边界数据包的处理。如果接收到窗口最左侧的数据包，接收端会检查并交付所有连续接收的数据包，然后适当地向前滑动窗口。这种机制允许接收

端按顺序交付数据，即使数据包是乱序到达的。窗口管理的优势在于它使得接收端能够有效地处理失序到达的数据包，并确保数据能按顺序交付给上层应用，同时还提供了一定程度的流控制。

4. 数据包的缓存与交付

在 SR 协议中，接收端可能会接收到乱序的数据包。为了保证数据能按正确的顺序交付给上层应用，接收端需要缓存这些失序的数据包，并在窗口内数据包连续时一起交付。

```

1  if ((seqNum >= rcv_base) && (seqNum < rcv_base + WindowSizeSize)) {
2      if (!isReceived[bufferIndex]) {
3          memcpy(packetBuffer[bufferIndex], receiveBuffer + sizeof(packet),
4                  packet.datasize);
5          isReceived[bufferIndex] = true;
6          cout << "\033[36m[INFO]\033[0m数据包成功接收并缓存_ _序列号:_ _" <<
7              seqNum << endl;
8
9          if (seqNum == rcv_base) {
10             while (isReceived[rcv_base % (2 * WindowSizeSize)]) {
11                 memcpy(Message + totalDataLength, packetBuffer[rcv_base % (2
12                     * WindowSizeSize)], packet.datasize);
13                 totalDataLength += packet.datasize;
14                 isReceived[rcv_base % (2 * WindowSizeSize)] = false;
15                 rcv_base++;
16                 cout << "\033[35m[MOVING]\033[0m收到顺序到来的确认号_ _窗口
17                     滑动_ _从_ _" << rcv_base << "_ _到_ _" << rcv_base +
18                     WindowSizeSize << endl;
19             }
20         }
21     }
22     // 其他逻辑...
23 }

```

在这段代码中，接收端首先检查一个新接收的数据包是否在当前接收窗口内。如果是，并且该数据包之前未被接收过，则将其数据复制到缓冲区中进行缓存，并标记为已接收。此外，如果收到的数据包是接收窗口的左边界（即 `seqNum == rcv_base`），接收端将检查并交付所有连续的缓存数据包。这是通过检查并更新 `isReceived` 数组来实现的，该数组跟踪了每个数据包的接收状态。通过这种方式，接收端能够有效地管理失序到达的数据包，并保证数据能够按正确的顺序交付给上层应用。

(四) 丢包测试设计

这种丢包测试设计对于评估 SR 协议的鲁棒性至关重要。它模拟了现实世界中网络不稳定造成的数据包丢失情况，帮助开发者了解和改进协议在面对网络不确定性时的表现。通过这样的测试，可以验证协议是否有效地处理数据包丢失。`shouldDropPacket` 函数在上述代码中用于模拟网络环境中的数据包丢失情况。这个函数的目的是随机决定是否应该“丢弃”一个数据包，以此来模拟在真实网络环境中可能发生的数据丢包现象。

```

1  #define LOSS_RATE 0.1
2  random_device rd;

```

```

3 mt19937 gen(rd());
4 uniform_int_distribution<> dis(0, 99);
5
6 bool shouldDropPacket() {
7     return dis(gen) < (LOSS_RATE * 100);
8 }

```

函数的实现涉及以下几个关键部分：

1. 使用 `random_device` 和 `std::mt19937` 创建一个伪随机数生成器 (`gen`)，用于生成均匀分布的随机数。
2. `uniform_int_distribution<> dis(0, 99)` 定义了一个范围在 0 到 99 之间的均匀分布。
3. 函数通过比较生成的随机数与 `LOSS_RATE * 100` 的结果来决定是否丢包。
4. 如果随机数小于 `LOSS_RATE * 100`，则函数返回 `true`，表示应该模拟丢包。

1. 发送分组丢包

发送分组丢包测试设计用于模拟在数据传输过程中数据包的丢失。这是通过在发送数据包前随机决定是否真正发送该包来实现的。

```

1 if (shouldDropPacket()) {
2     cout << "[WARNING] 分组丢包测试 - 丢失: " << nextSeqNum << "号数据包"
3     << endl;
4 }
5 else {
6     sendto(socketClient, stagingBuffer[nextSeqNum % WindowSize], sizeof(
7         packet) + packetDataSize, 0, (sockaddr*)&servAddr, servAddrlen);
8     cout << "\033[33m[SEND]\033[0m 发送分组 - 序列号: " << nextSeqNum << ",
9         packet 序列号: " << int(packet.seq) << ", 校验和: " << int(packet.
10        checksum) << " " << endl;
11 }

```

在这个片段中，`shouldDropPacket` 函数根据定义的丢包率 (`LOSS_RATE`) 随机决定是否丢弃一个特定的数据包。如果函数返回 `true`，则模拟丢包，不发送当前数据包，并打印相应的警告信息。如果返回 `false`，则正常发送数据包。

2. 接收确认 (ACK) 丢包

接收确认丢包测试设计用于模拟在 ACK 传输过程中的丢包情况。这可以帮助评估协议如何处理丢失的 ACK，以及这会如何影响整个通信过程。

```

1 if (shouldDropPacket()) {
2     cout << "[WARNING] ACK丢包测试 - 丢失: " << seqNum << "号数据包的确认回
3     复" << endl;
4 }
5 else {
6     sendto(socketServer, receiveBuffer, sizeof(ackPacket), 0, (sockaddr*)&
7         clieAddr, clieAddrlen);
8 }

```

```
6     cout << "\033[33m[SEND]\033[0m 发送确认信息 - 确认号:" << int(ackPacket.  
7     ack) << ", 校验和:" << int(packet.checksum) << endl;  
    }
```

在这个片段中，发送 ACK 之前也会通过 `shouldDropPacket` 函数来决定是否实际发送 ACK。如果函数返回 `true`，则模拟 ACK 丢包，不发送 ACK，并打印相应的警告信息。如果返回 `false`，则正常发送 ACK。

(五) 其他部分

- 建立连接（三次握手）
- 断开连接（四次挥手）

（详见 3-1）

(六) 补充说明：窗口大小与序列号范围的关系

在 SR 协议中，每个数据包都有一个唯一的序列号，这些序列号是从一个固定大小的序号空间中分配的。例如，如果序列号是一个字节长，则序号空间大小为 256（序列号范围从 0 到 255）。窗口长度定义了接收方在任何给定时刻可以接收或期待的数据包数量。为了确保协议的正确性，窗口长度必须设计得足够小，以避免序列号的重复或混淆。如果窗口长度超过序号空间大小的一半，可能会出现一个新到达的数据包的序列号与已经被接收和处理的旧数据包的序列号相同的情况。在这种情况下，接收方无法确定收到的数据包是新的还是旧的重复数据包。

为了避免这种混淆，窗口长度必须设计为小于或等于序号空间大小的一半。这样，即使序列号循环使用，接收方也能够根据其窗口位置准确地识别数据包。

四、 协议优化部分

(一) 问题指出

在选择重传（Selective Repeat, SR）协议中，发送方和接收方各自维护各自的滑动窗口，用于控制可以发送或接收的数据包序列号的范围。然而，接收端窗口大小必须设置为发送端窗口大小的两倍是基于协议效率和可靠性的关键考虑。以下是对这一设计决策的详细解释。（注意：本部分在于蔡学长讨论以后，我认识到这部分的优化并非绝对必要。在选择重传协议中，接收端针对每个分组发送的确认（ACK）是基于各自的序列号的。这意味着，即使接收端的窗口已经滑动并且之前接收到的分组确认信息丢失，接收端依旧能够根据相应的序列号向发送端回复相应的 ACK。因此，对窗口大小的这种优化可能并不是必需的。感谢蔡学长提供的指导，这有助于我更准确地理解 SR 协议的工作机制。）

- 在某些极端情况下，接收方的窗口可能会完全领先于发送方的窗口。这一情况发生在所有发送方窗口内的数据包都被接收方接收，但相应的确认（ACK）还未被发送方收到时。举一个具体的例子：假设发送方和接收方的窗口大小均为 4（ $N=4$ ）。发送方发送窗口内的所有数据包（序列号 0, 1, 2, 3），而接收方也成功接收这些数据包并发送相应的 ACK。然而，这些 ACK 在传输过程中全部丢失，导致发送方未收到任何确认并保持其窗口不变。与此同时，接收方根据已接收的数据包移动其窗口，现在覆盖新的序列号 4, 5, 6, 7。当发送方的定时器超时，它开始重传序列号 0, 1, 2, 3 的数据包。由于接收方的窗口实际处理范围是

2N (即 0 到 7), 它仍然能够接收并处理这些重传的数据包, 并再次发送 ACK。这次 ACK 成功到达发送方后, 发送方更新其窗口为新的序列号 4, 5, 6, 7。反之, 如果接收方的窗口实际处理范围是 N, 此时由于窗口已经划过, 则接收端无法再次向发送端发送接收确认信息

- 如果接收方只处理其滑动窗口内的数据包, 则无法再次发送对于那些已经被接收但 ACK 未被发送方接收的包的确认。这可能导致发送方重复发送这些数据包。
- 通过处理 $rcv_base - N$ 到 $rcv_base - 1$ 区间内的数据包, 接收方可以对这些包再次发送 ACK, 通知发送方这些数据包已经被成功接收, 防止不必要的重传。
- 在接收方窗口大小为 2N 的情况下, 即使在某些数据包的 ACK 丢失的情况下, 发送方窗口也能继续向前滑动, 维护数据传输的连续性和效率。

(二) 优化实现

在本次实验设计的过程中, 上述问题引起了我的强烈关注。需要指出, 本次实验的要求中的“发送窗口和接收窗口采用相同大小”在具体实践中是不正确的。为此在本次实验中, 我针对此问题进行了改正, 将接收端窗口大小必须设置为发送端窗口大小的两倍, 具体的代码实现参照前文“接收端设计”部分的内容, 完整代码详见压缩包中 server.cpp 文件。

五、 优点与不足

(一) 优点

首先, SR 协议允许独立确认每个数据包, 这意味着即使某些包丢失或延迟, 其它已发送的包仍可被确认。这减少了不必要的重传, 从而更高效地利用带宽。在高误码率的网络环境下, SR 协议由于避免了不必要的重传, 能够提供更好的网络性能。其次, SR 协议能够快速识别和重传丢失的数据包, 从而减少整体传输延迟。同时, SR 协议的窗口滑动机制比 Go-Back-N (GBN) 协议更加灵活, 因为它允许发送方在收到非连续的 ACK 后继续发送数据。最后, SR 协议能够适应网络条件的变化, 如拥塞时减慢发送速率, 从而减少网络中的拥塞情况。

(二) 不足

相比于 Go-Back-N 等其它协议, SR 协议的实现更为复杂, 特别是在处理序列号、计时器和缓冲区管理方面。同时, SR 协议要求发送方和接收方都维护更大的缓冲区来存储数据包, 这可能导致较高的内存需求。在网络条件良好时, SR 协议的某些特性 (如为每个包维护独立定时器) 可能导致资源的不必要使用。其次, SR 协议的效率在很大程度上依赖于定时器的准确性。不准确的定时器可能导致不必要的重传或延迟确认。最后, 尽管 SR 协议能够处理乱序到达的包, 但如何正确处理和重组这些包是需要精心设计的, 实现起来相对复杂。

六、 结果展示

(一) 实验结果

The screenshot shows a network simulation window titled 'C:\Users\ZhengYafu\Desktop'. It displays a series of log messages indicating the successful transmission of a file named 'helloworld.txt'. The messages include sequence numbers, packet numbers, and checksums. The file size is 145888 bytes. The transmission is successful, and the file is saved.

(a) helloworld.txt

The screenshot shows a network simulation window titled 'C:\Users\ZhengYafu\Desktop'. It displays a series of log messages indicating the successful transmission of a file named '1.jpg'. The messages include sequence numbers, packet numbers, and checksums. The file size is 157333 bytes. The transmission is successful, and the file is saved.

(b) 1.jpg

The screenshot shows a network simulation window titled 'C:\Users\ZhengYafu\Desktop'. It displays a series of log messages indicating the successful transmission of a file named '2.jpg'. The messages include sequence numbers, packet numbers, and checksums. The file size is 949888 bytes. The transmission is successful, and the file is saved.

(c) 2.jpg

The screenshot shows a network simulation window titled 'C:\Users\ZhengYafu\Desktop'. It displays a series of log messages indicating the successful transmission of a file named '3.jpg'. The messages include sequence numbers, packet numbers, and checksums. The file size is 1136899 bytes. The transmission is successful, and the file is saved.

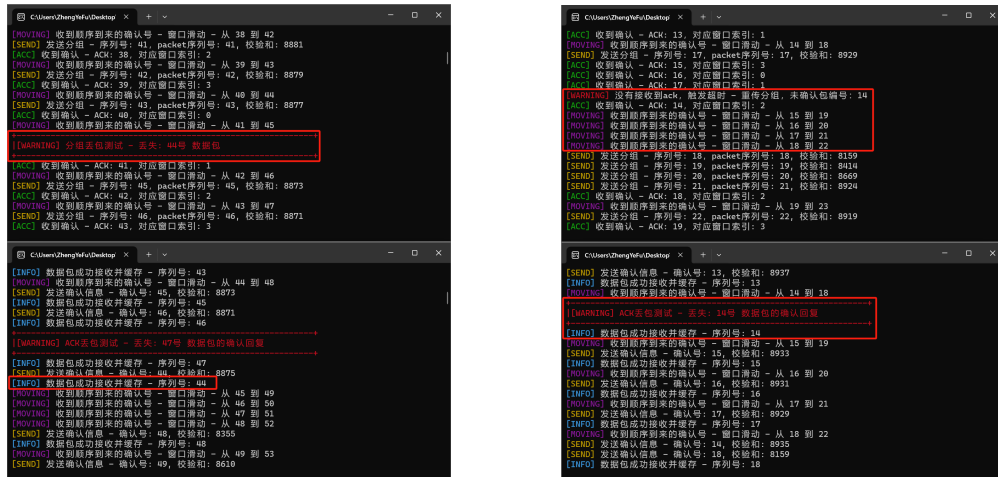
(d) 3.jpg

(二) 三次握手

The screenshot shows a network simulation window titled 'C:\Users\ZhengYafu\Desktop'. It displays a series of log messages indicating the successful establishment of a connection through a three-way handshake. The messages include sequence numbers, packet numbers, and checksums. The connection is successful, and the file is saved.

图 2: 三次握手

(三) 丢包测试

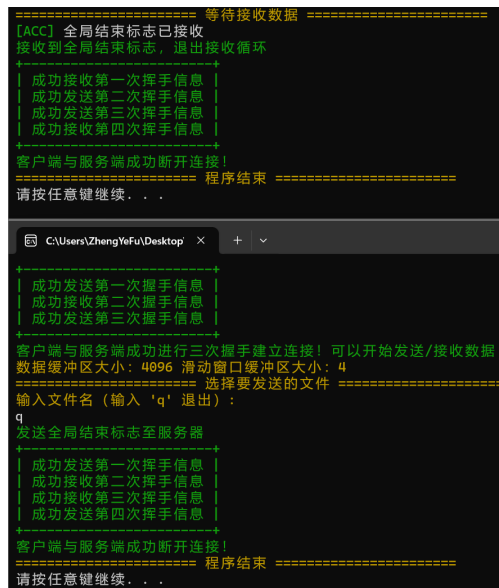


The image contains two screenshots of a network simulation interface. The left screenshot, labeled (a), shows a '分组丢包测试' (Packet Loss Test) where a warning message indicates a loss of packet 44. The right screenshot, labeled (b), shows a '接收确认 (ACK) 丢包测试' (ACK Loss Test) where a warning message indicates a loss of ACK 14. Both screenshots show a series of log messages for sending and receiving packets and ACKs, with sequence numbers and checksums.

(a) 分组丢包测试

(b) 接收确认 (ACK) 丢包测试

(四) 四次挥手



The image contains two screenshots of a terminal window showing the '四次挥手' (Four-Way Handshake) process. The top screenshot shows the initial state where the client and server are successfully connected. The bottom screenshot shows the process of sending and receiving the four挥手 (handshakes) to terminate the connection. The terminal output includes messages like '成功接收第一次挥手信息', '成功发送第二次挥手信息', etc., and a final message '客户端与服务端成功断开连接!'. The terminal also shows the buffer sizes and the file name 'q' being used for the test.

图 4: 四次挥手

七、 实验心得与体会

本次实验对选择重传 (Selective Repeat, SR) 协议的实现过程提供了深刻的见解, 尤其在窗口大小设置方面的探索, 让我对协议的内部工作原理有了更加深入的理解。实验中, 我特别关注了发送窗口和接收窗口大小的不匹配问题, 并根据协议的需求和网络环境的特点, 将接收窗口大小设置为发送窗口大小的两倍。这一改变不仅解决了潜在的数据包识别问题, 还优化了数据传输的整体效率和可靠性。在实验过程中, 我深刻体会到理论知识与实践应用之间的差异。课堂上学到的理论和标准实现在面对实际网络环境时往往需要调整。特别是在设计和实现网络协议时, 理论上合理的假设可能在实际应用中遇到挑战。例如, 原本协议中发送和接收窗口大小相同的设

计，在实际测试中显示出了局限性，这迫使我重新思考并调整窗口大小，以适应可能的极端网络条件。

通过这次实验，我也更加明白了网络协议设计中灵活性和鲁棒性的重要性。特别是在处理丢包和乱序到达的数据包时，一个有效的协议应该能够适应各种网络状况，保持数据传输的连续性和可靠性。SR 协议中增加的接收窗口范围正是为了应对这些情况，确保即使在 ACK 丢失或数据包乱序到达的情况下，也能有效地继续数据传输。

参考文献