南開大學

计算机网络实验报告

Lab3-2

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

在本次实验中，我们采用了滑动窗口机制，代替了原有的停等机制，以提升数据传输的效率和可靠性。报文格式定义了窗口字段，这个字段指示了滑动窗口的可用空间大小。在流水线协议（GBN）中，每个数据包被分配了一个唯一的序列号，而发送和接收方都维护一个缓冲区。发送方可以连续发送多个数据包，但数量不超过窗口大小。如果在预定时间内未收到确认，将触发超时重传。接收方仅接受按顺序到达的数据包，如果收到非预期的包，将丢弃并重发上一个已确认的包的确认。累积确认的机制允许接收方对按序到达的最后一个包发送确认。超时重传机制通过计时器控制，未在超时时间内收到确认的数据包将被重传。

关键字：滑动窗口机制；　Go-Back-N 流水线协议；　累积确认机制

目录

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，支持累积确认，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
- 累积确认：Go Back N
- 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、 协议设计

(一) 报文格式

```
typedef struct Packet_Header
{
    BYTE window;
    ...
}
```

- 在 TCP 协议的头部中，第 24 到 31 位被标记为窗口（window）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。
- 其余部分详见实验 3-1

(二) 流水线协议（GBN）& 发送/接收缓冲区 & 累积确认

1. 原理介绍

1. **序列号**:
   - 每个数据包都被分配一个唯一的序列号。
   - 序列号的范围通常是 $[0, 2^n - 1]$，其中 n 是序列号字段的位数。
2. **发送方行为**:
   - 发送方可以连续发送多个数据包，只要它们的数量不超过窗口大小。
   - 发送的每个包必须在预定的时间内收到确认，否则将触发超时重传。
3. **接收方行为**:

- 接收方只接受按顺序的数据包。如果收到一个序列号不是预期的包，它将丢弃该包并重发上一个已确认的包的确认。

4. **发送/接收缓冲区**：
   - 发送方和接收方都维护一个缓冲区。
   - 发送缓冲区大小限制了在不需要等待确认的情况下可以发送的最大数据包数量。
   - 接收缓冲区通常大小为 1，表示接收方一次只能接受一个按顺序的包。
5. **累积确认**:
   - 接收方对按序到达的最后一个包发送确认。
   - 确认包包含下一个期望接收的包的序列号。
   - 如果发送方没有在设定的超时时间内接收到确认，它将重传当前窗口中所有未被确认的数据包。

2. 代码实现

(1) 流量控制机制——滑动窗口

发送方 (SendMessage):

- `Window` 参数定义了滑动窗口的大小。
- `currentPacketIndex` 和 `lastUnacknowledgedPacketIndex` 用于跟踪窗口内的数据包。
- 当发送窗口未满时，代码连续发送数据包，并更新窗口状态。

```
while (currentPacketIndex <= lastConfirmedPacketIndex + Window &&
       currentPacketIndex < totalPacketCount) {
        int packetDataSize = (currentPacketIndex == totalPacketCount - 1 ?
            mes_size - (totalPacketCount - 1) * MAX_SIZE : MAX_SIZE);

        packet.tag = 0;
        packet.seq = Seq_num++;
        Seq_num = (Seq_num > 255 ? Seq_num - 256 : Seq_num);
        packet.datasize = packetDataSize;
        stagingBufferLengths[currentPacketIndex % Window] = packetDataSize;
        packet.window = Window - (currentPacketIndex -
            lastConfirmedPacketIndex);/
        packet.checksum = 0;
        packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));

        memcpy(stagingBuffer[currentPacketIndex % Window], &packet, sizeof(
            packet));
        char* messageFragment = Message + currentPacketIndex * MAX_SIZE;
        memcpy(stagingBuffer[currentPacketIndex % Window] + sizeof(packet),
            messageFragment, packetDataSize);

```

```
18          sendto(socketClient, stagingBuffer[currentPacketIndex % Window],
                sizeof(packet) + packetDataSize, 0, (sockaddr*)&servAddr,
                servAddrlen);
19
20          if (!timerRunning || lastUnacknowledgedPacketIndex ==
                currentPacketIndex) {
21              timerStart = clock();
22              timerRunning = true;
23          }
24          currentPacketIndex++;
25      }
26  }
```

接收方 (RecvMessage):

- 接收方通过检查序列号来接收和处理数据包。
- 若序列号与预期不符，接收方请求重传。

```
1   // 如果接收到的序列号不是预期的, 发送重传请求
2   if (packet.seq != Seq_num) {
3       Packet_Header resendHeader;
4       resendHeader.tag = 0;
5       resendHeader.ack = Ack_num - 1;  // 发送上一个确认的ACK
6       resendHeader.checksum = 0;
7       resendHeader.checksum = compute_sum((WORD*)&resendHeader, sizeof(
            resendHeader));
8       memcpy(receiveBuffer, &resendHeader, sizeof(resendHeader));
9       sendto(socketServer, receiveBuffer, sizeof(resendHeader), 0, (sockaddr*)&
            clieAddr, clieAddrlen);
10      continue;
11  }
12
13  // 成功接收数据
14  singleDataLength = packet.datasize;
15  memcpy(Message + totalDataLength, receiveBuffer + sizeof(packet),
        singleDataLength);
16  totalDataLength += singleDataLength;
17
18  // 确认接收成功, 发送ACK
19  packet.tag = 0;
20  packet.ack = Ack_num++;
21  packet.seq = Seq_num++;
22  packet.datasize = 0;
23  packet.checksum = 0;
24  packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
25  memcpy(receiveBuffer, &packet, sizeof(packet));
26  sendto(socketServer, receiveBuffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
        clieAddrlen);
```

(2) 数据包对应多个序列号

- 在两个函数中，`Seq_num` 变量用于跟踪数据包的序列号。
- 发送方为每个数据包分配唯一的序列号，而接收方根据这些序列号来识别和确认数据包。

```
packet.seq = Seq_num++;
Seq_num = (Seq_num > 255 ? Seq_num - 256 : Seq_num);
```

(3) 确认机制——累计确认

发送方 (`SendMessage`):

- 收到的确认（ACK）用于更新滑动窗口。
- `Ack_num` 变量用于追踪期望的 ACK 序号。
- 收到 ACK 后，确认对应的数据包并移动窗口。

```
if (packet.ack == Ack_num) {
    Ack_num = (Ack_num + 1) % 256;
    lastConfirmedPacketIndex++;
    lastUnacknowledgedPacketIndex = (lastUnacknowledgedPacketIndex + 1) %
        Window;
    if (lastUnacknowledgedPacketIndex == currentPacketIndex) {
        timerRunning = false;
    }
}
else {
    // 处理1:检测到重复确认
    if (packet.ack == (Ack_num == 0 ? 255 : Ack_num - 1)) {
        coutinue;
    }

    // 处理2:所有累积确认的数据包
    else if (dis < Window || (Ack_num + dis) % 256 == packet.ack) {
    while (Ack_num != (packet.ack + 1) % 256) {
        lastConfirmedPacketIndex++;
        lastUnacknowledgedPacketIndex = (lastUnacknowledgedPacketIndex + 1) %
            Window;
        Ack_num = (Ack_num + 1) % 256;
    }

    // 处理3:重发所有未确认的数据包
     else {
         for (int temp = lastConfirmedPacketIndex + 1; temp <=
             lastConfirmedPacketIndex + Window && temp < totalPacketCount;
             temp++) {
```

```
            sendto(socketClient, stagingBuffer[temp % Window], sizeof(packet
                ) + stagingBufferLengths[temp % Window], 0, (sockaddr*)&
                servAddr, servAddrlen);
        }
    }
```

接收方 (RecvMessage):

- 收到预期序列号的数据包后，接收方发送 ACK。
- 使用 `Ack_num` 变量跟踪并发送下一个预期的 ACK。

```
// 确认接收成功, 发送ACK
packet.ack = Ack_num++;
...
memcpy(receiveBuffer, &packet, sizeof(packet));
sendto(socketServer, receiveBuffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
    clieAddrlen);
```

(4) 超时重传

发送方 (SendMessage):

- 使用 `timerStart` 和 `timerRunning` 变量来实现超时机制。
- 如果在超时时间内未收到确认，将重传窗口内的所有数据包。
- 超时时间通过计时器和系统时钟来控制。

```
if (timerRunning && (clock() - timerStart) / CLOCKS_PER_SEC > 2) {
    for (int temp = lastUnacknowledgedPacketIndex; temp < currentPacketIndex;
         temp++) {
        sendto(socketClient, stagingBuffer[temp % Window], sizeof(packet) +
            stagingBufferLengths[temp % Window], 0, (sockaddr*)&servAddr,
            servAddrlen);
    }
    timerStart = clock();
}
```

(5) 数据校验和重传请求

- 代码中通过 `compute_sum` 函数来计算和验证数据包的校验和。
- 如果接收到的数据包校验失败，接收方会请求重传。

```
if (packet.tag == 0 && (compute_sum((WORD*)&packet, sizeof(packet)) == 0)) {
    // 如果接收到的序列号不是预期的, 发送重传请求
    if (packet.seq != Seq_num) {
        Packet_Header resendHeader;
```

```
        resendHeader.tag = 0;
        resendHeader.ack = Ack_num - 1;  // 发送上一个确认的ACK
        resendHeader.checksum = 0;
        resendHeader.checksum = compute_sum((WORD*)&resendHeader, sizeof(
            resendHeader));
        memcpy(receiveBuffer, &resendHeader, sizeof(resendHeader));
        sendto(socketServer, receiveBuffer, sizeof(resendHeader), 0, (
            sockaddr*)&clieAddr, clieAddrlen);
        continue;
    }
    // 成功接收数据...
}
```

(6) 发送端和接收端交互

- 在 `SendMessage` 中，发送方基于接收到的 ACK 更新其窗口和状态。
- 在 `RecvMessage` 中，接收方根据收到的数据包的内容更新其内部状态。

```
// SendMessage 基于接收到的ACK更新其窗口和状态。
if (packet.ack == Ack_num) {
    Ack_num = (Ack_num + 1) % 256;
    lastConfirmedPacketIndex++;
    lastUnacknowledgedPacketIndex = (lastUnacknowledgedPacketIndex + 1) %
        Window;
    if (lastUnacknowledgedPacketIndex == currentPacketIndex) {
        timerRunning = false;
    }
}

// RecvMessage 根据收到的数据包的内容更新其内部状态。
// 成功接收数据
singleDataLength = packet.datasize;
memcpy(Message + totalDataLength, receiveBuffer + sizeof(packet),
    singleDataLength);
totalDataLength += singleDataLength;

// 确认接收成功, 发送ACK
packet.tag = 0;
packet.ack = Ack_num++;
packet.seq = Seq_num++;
packet.datasize = 0;
packet.checksum = 0;
packet.checksum = compute_sum((WORD*)&packet, sizeof(packet));
sendto(socketServer, receiveBuffer, sizeof(packet), 0, (sockaddr*)&clieAddr,
    clieAddrlen);
```

3. 优点与不足

(1) 优点

Go-Back-N (GBN) 流水线机制的主要优点在于其简单性和高效性。在网络错误率较低的环境中，GBN 能够有效地利用网络带宽，通过发送多个数据包而无需等待每个包的确认来提高传输效率。此外，GBN 提供了可靠的数据传输机制，通过自动重传丢失或错误的数据包来确保数据的完整性和准确性。

(2) 缺点

在高延迟或高丢包率的网络环境中，它的带宽利用率可能会降低。这是因为在发生丢包或错误时，GBN 需要重传整个窗口内的所有包，而不仅仅是丢失或错误的包，这会导致大量的冗余数据传输。另外，由于接收方仅接受按顺序到达的数据包并丢弃所有乱序到达的包，这可能导致在包乱序而不是丢失的情况下出现不必要的重传。

(三) 优化部分

1. 快速重传

下述代码实现了快速重传机制。它通过监测重复确认（ACK）信号来判断数据包是否丢失。一旦收到三次重复的确认，就会立即重传最早未被确认的数据包，而不是等待重传计时器超时。这样做可以有效减少因等待而产生的延迟，提高数据传输的效率。

```
int duplicateAckCount = 0;
// 检测到重复确认
if (packet.ack == (Ack_num == 0 ? 255 : Ack_num - 1)) {
    // 如果Ack_num是0，前一个确认号应该是255，否则就是Ack_num - 1。
    cout << "\033[31m[WARNING]\033[0m␣收到重复确认，Ack:␣" << int(packet.ack)
         << endl;
    // 快速重传机制
    duplicateAckCount++;
    if (duplicateAckCount >= 3) {
        // 重传最早未被确认的数据包
        int packetIndex = lastUnacknowledgedPacketIndex % Window;
        sendto(socketClient, stagingBuffer[packetIndex], sizeof(packet) +
            stagingBufferLengths[packetIndex], 0, (sockaddr*)&servAddr,
            servAddrlen);
        cout << "\033[31m[INFO]\033[0m␣快速重传触发，重传数据包␣Seq:␣" <<
            lastUnacknowledgedPacketIndex << endl;

        // 重置重复ACK计数
        duplicateAckCount = 0;
    }
}
```

(四) 其他部分

- 建立连接（三次握手）

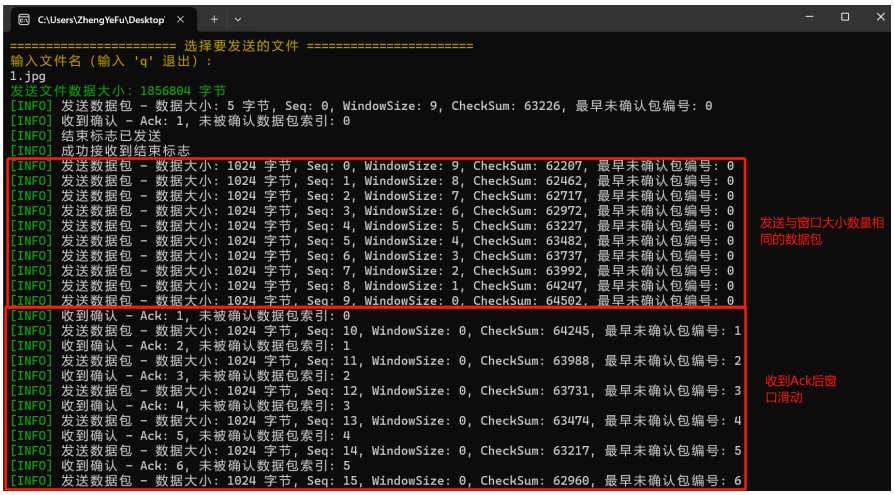
- 断开连接（四次挥手）

  详见 3-1

三、 结果展示

1. **建立连接 & 窗口大小**

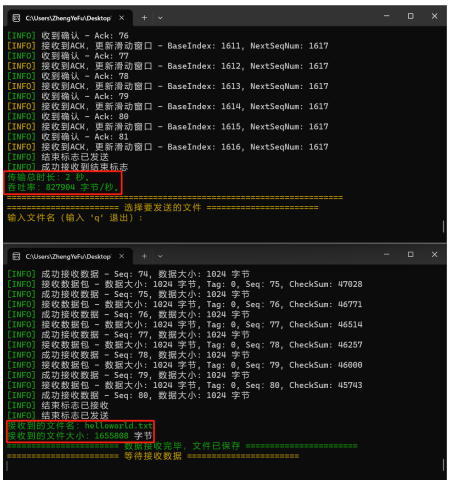
2. **发送方日志（序号 & ACK& 校验和）**



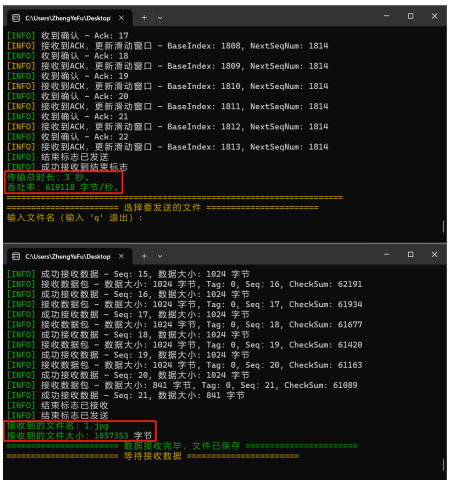
3. **接收方日志（序号 & ACK& 校验和）**

4. **断开连接**

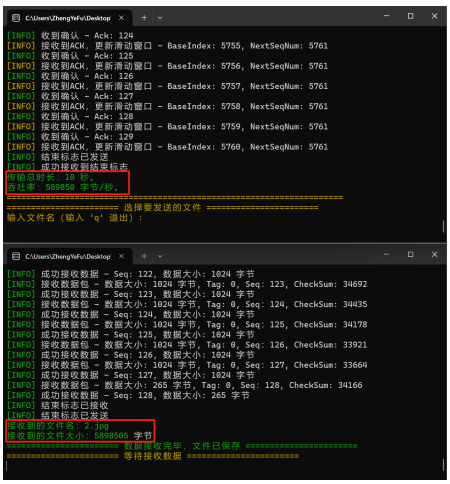
5. **时延与吞吐率**



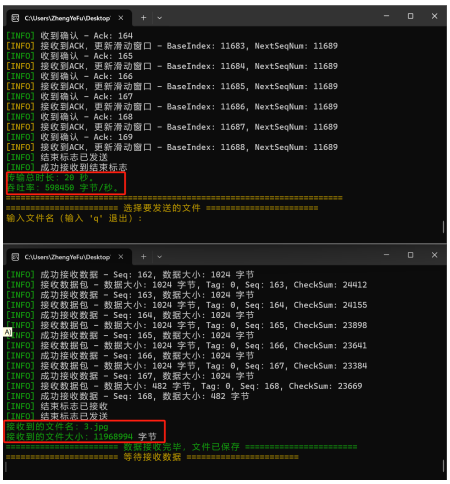
(a) helloworld.txt



(b) 1.jpg

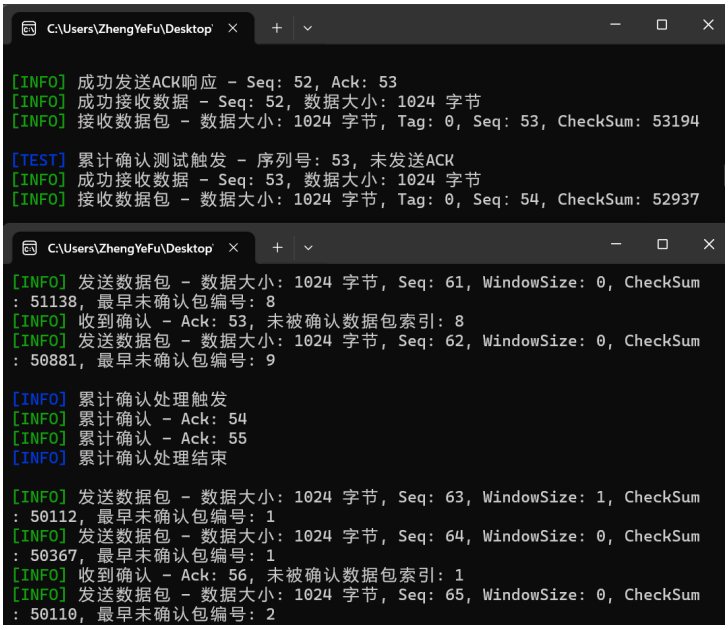


(c) 2.jpg

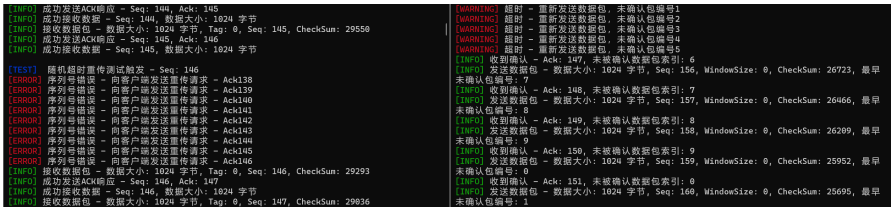


(d) 3.jpg

6. 累计确认机制



7. **超时重传机制**



四、 实验心得与体会

Go-Back-N (GBN) 流水线机制的引入提升了本次实验的效率和可靠性。在网络错误率较低的环境中，GBN 显著提高了网络带宽的利用率，允许发送多个数据包而无需等待每个包的确认。这种机制在保障数据完整性和准确性的同时，也实现了高效的数据传输。然而，在高延迟或高丢包率的网络环境中，GBN 的带宽利用率可能会降低，因为丢包或错误会导致整个窗口内所有包的重传，造成大量冗余数据传输。此外，由于接收方只接受按顺序到达的数据包，这可能导致在包乱序而不是丢失的情况下出现不必要的重传。整体来看，GBN 流水线机制在实现高效数据传输的同时，也暴露了在特定网络条件下的局限性。

参考文献