南开大学

Nankai University

南　　开　　大　　学

计　算　机　学　院

计算机网络实验报告

计算机网络实验 3-2

张烨

年级：2022 级

学号：2212108

专业：计算机科学与技术

指导教师：吴英

2024 年 12 月 5 日

摘要

关键字：数据包套接字、基于滑动窗口的流量控制、GBN、累积确认

目录

一、 协议设计 & 核心代码分析

(一) 数据包格式、建立连接、差错检测——沿用 lab3-1

数据包格式与建立连接、差错检测的功能沿用于 lab3-1, 此处简单叙述，不再放上代码赘述。

1. 数据包格式

数据包格式由两个主要部分组成：伪首部（‘PseudoHeader‘）和数据报（‘Datagram‘）。

伪首部包含了进行 UDP 校验和计算所需的关键信息。它包括源 IP 地址和目的 IP 地址（各为 32 位），一个 8 位的零字段作为填充，协议号字段（UDP 协议的值为 17，8 位），以及 UDP 数据报的长度（16 位）。伪首部的定义主要是服务于差错检测。

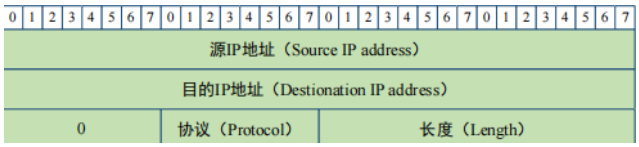


图 1: 伪首部设计

数据报则是实际的数据包结构，主要组成部分包括源端口号、目的端口号、序列号和确认号（各为 16 位），当前数据大小（16 位），校验和（16 位），以及标志位（16 位），该标志位用于指示数据包类型，如数据、SYN、SYN-ACK、ACK、FIN 和 FIN-ACK。最后，数据部分是一个字符数组，大小为 ‘BUFFER_SIZE‘，用于存储实际的数据内容。

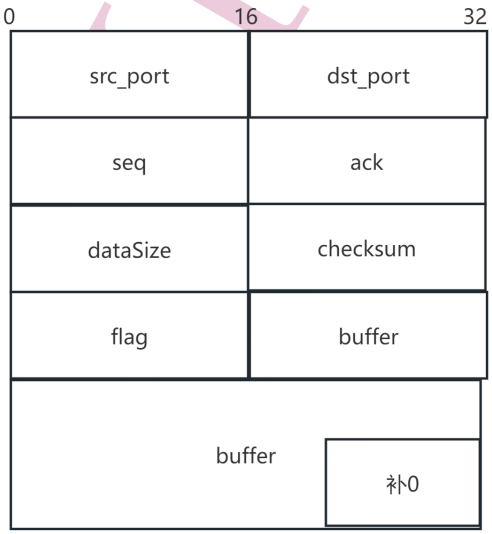


图 2: 数据包 Datagram 设计

2. 建立与断开连接

客户端与服务器端的建立连接过程，模仿了 TCP 的三次握手；断开连接过程则简化了 TCP 的四次挥手，改为三次挥手。此处的设计，由客户端主动发起连接、主动断开连接。

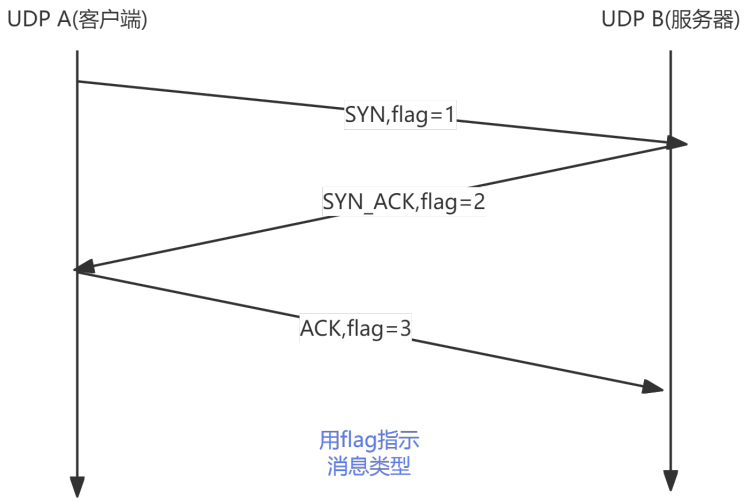


图 3: 建立连接：模拟 TCP 三次握手过程

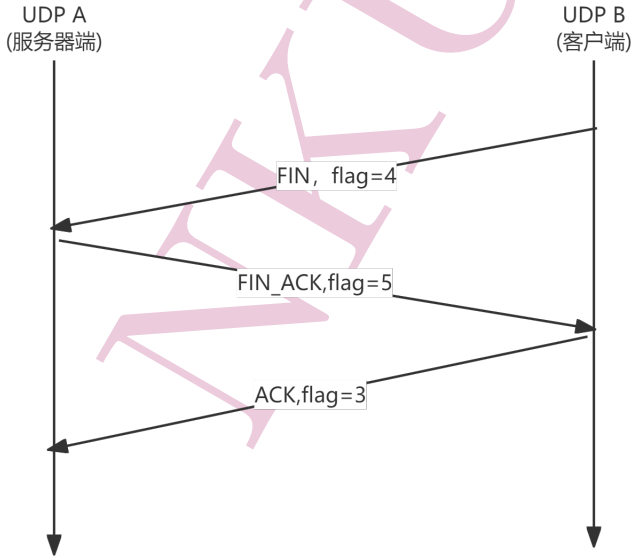


图 4: 断开连接：简化 TCP 四次挥手为三次挥手

3. 差错检测

为了确保在数据传输过程中数据的完整性，我设计的协议使用校验和（checksum）机制来检测数据包是否发生了损坏，主要分为计算和验证两个步骤。

在“计算校验和”的过程中，首先构建一个伪首部（‘PseudoHeader‘），该首部包含源地址、目的地址、协议类型（UDP）和数据长度等信息。伪首部与数据报结构体（‘Datagram‘）的数据部分一起被复制到一个缓冲区中。接着，通过一个循环将缓冲区中的数据以 16 位为单位进行求和，如果在求和过程中产生了进位，则将进位部分加回到和中。最终，计算出的和经过取反后生成校验和，并用于数据报中。

计算过程可以归纳如下:

- 产生伪首部，校验和域段清 0，将数据报用 0 补齐为 16 位整数倍
- 将伪首部和数据报一起看成 16 位整数序列
- 进行 16 位二进制反码求和运算，计算结果取反写入校验和域段

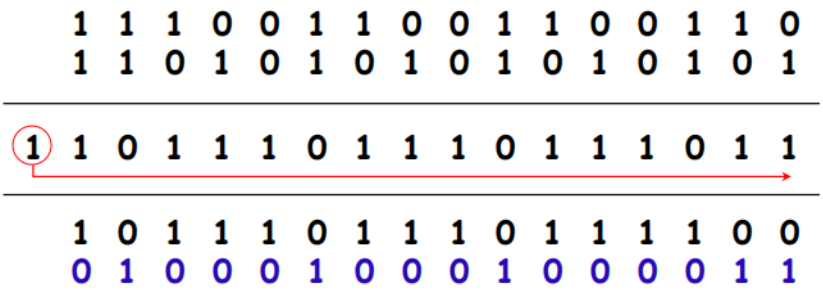


图 5: 校验和计算示例。伪首部与数据报结构体的数据以 16 位为单位进行取反、求和，如果在求和过程中产生了进位，则将进位部分加回到和中。最终，计算出的和经过取反后生成校验和。

在" 验证校验和" 时，程序会重新计算数据包的校验和并与存储的校验和进行比较。如果两者相等，则说明数据包在传输过程中的完整性得到保证，否则数据包被认为发生了错误。

(二) 基于滑动窗口的流量控制

1. 发送端 (客户端)

对于发送端，发送窗口为 N>1，每次发送的基准窗口设置为 base（即每轮发送的最小序列值)。并且:

- 允许发出 N 个未得到确认的分组，连续发送而不等待接收端的回复
- 如果在定时范围内收到该轮最大的 ack(expectedAck=base+N-1), 立即更新基准窗口 base=expectedAck+1, 并且开始下一轮发送。(也就是说，即使定时器时间未到，也立刻开始下一轮发送)
- 如果在定时范围内没有收到该轮最大的 ack，将 base 更新为这段时间收到的最大 ack 值 +1, 即 base=max(getAck)+1. 并且从 base 开始进行下一轮发送。

下面展示流量控制涉及的两个关键函数，代码之后附有图示进行解释。

客户端监听 ACK

```
1  void Sender::receiveAck() {
2      Datagram ackPacket(SERVER_PORT,ROUTER_PORT);
3      socklen_t len = sizeof(routerAddr);
4      while (true) {
5          if (recvfrom(sock, reinterpret_cast<char*>(&ackPacket), sizeof(
               ackPacket), 0, (struct sockaddr*)&routerAddr, &len) > 0) {
6              if (ackPacket.flag == 0 && ackPacket.validateChecksum(clientAddr.
                   sin_addr.S_un.S_addr, routerAddr.sin_addr.S_un.S_addr)) {
7                  if(ackPacket.ack==65535)
```

```
                {
                    continue;
                }
                std::lock_guard<std::mutex> lock(mtx);
                std::cout << "收到ack=" << ackPacket.ack << std::endl;
                if(ackPacket.ack>getAck){getAck=ackPacket.ack;}
                if(ackPacket.ack==expectedAck)
                {
                    ackReceived = true;
                    cv.notify_one();
                }
            }
        }
    }
}
```

客户端超时后发送或收到最大 ACK 后提前发送

```
bool Sender::waitForAck() {
    std::unique_lock<std::mutex> lock(mtx);
    //如果 ackReceived 在超时之前变为 true，则 wait_for 返回并继续执行后续代码;
    //如果超时后 ackReceived 仍为 false，则 wait_for 也会返回。
    return cv.wait_for(lock, std::chrono::milliseconds(5*TIMEOUT), [this]() {
        return ackReceived.load() ; });//超时或者收到所有ack返回
}
```

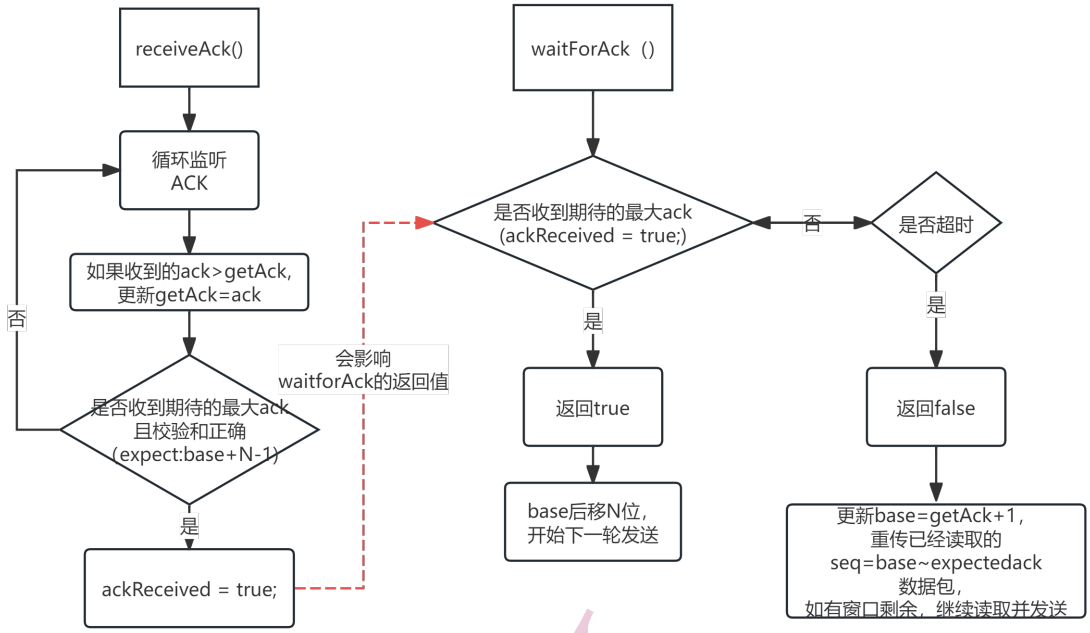


图 6: 监听 ack, 更新滑动窗口（基准值为 base）的示例图。客户端每收到一个 ack, 就会比较 ack 与 getACK 的值，把最大 ack 赋给 getACK. 如果客户端收到了期待的最大 ack(即传输过程中没有出现丢包)，会立即更新 base 并开始下一轮发送；如果客户端未收到期待的最大 ack，会陷入等待，定时器到时后会将 base 更新为 getAck+1, 已经读取过的数据包重新发送、未读取的数据包读取后再发送。

下面则是文件发送的主体代码。由于 base,ackReceived 变量在多个线程中都可能被访问，添加了“std::unique_lock<std::mutex> lock(mtx)”来确保变量值的正确更新与读取。

可以看到，文件会连续读取序列号位于 [nextseq,base+WINDOW_SIZE] 之间的数据包，并连续发送给服务器端。如果客户端收到了期待的最大 ack，会立即更新 base=base+N 并开始下一轮发送; 如果客户端未收到期待的最大 ack，会陷入等待，定时器到时后会将 base 更新为 getAck+1, 已经读取过的数据包重新发送、未读取的数据包读取后再发送。

客户端基于滑动窗口的文件发送

```
1  void Sender::sendFile(const std::string& filename) {
2      //.......
3          base = 0;
4          nextseq = 0;
5          while (true) {
6              //.......
7              //1.基于窗口连续发送N条消息
8              while (nextseq < base + WINDOW_SIZE && !file.eof()) {
9                  if(nextseq<3&&AckPacket.flag == 2&&AckPacket.validateChecksum
                       (clientAddr.sin_addr.S_un.S_addr, routerAddr.sin_addr.
                       S_un.S_addr))//2.如果此时又收到了SYN-ACK
10                 {
11                     //...重新发送
12                 }
```

```
                std::unique_lock<std::mutex> lock(mtx);
                ackReceived = false;
                expectedAck = base+WINDOW_SIZE-1;
                Datagram packet;//默认构造，从客户端发往路由器端
                packet.seq = nextseq;
                file.read(packet.data, BUFFER_SIZE);
                packet.dataSize = static_cast<int>(file.gcount());
                packet.flag = 0;

                window[nextseq] = packet;
                //window.emplace(nextseq, packet);
                sendPacket(packet);
                std::cout << "发送数据包.SEQ=" << packet.seq <<", 校验码="<<
                    packet.checksum << std::endl;
                nextseq++;
                lock.unlock();
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(TIMEOUT));

            // 2.结束条件：如果所有包都被确认，则跳出
            if (base == nextseq && file.eof()) break;
            if (waitForAck()) {
                std::unique_lock<std::mutex> lock(mtx);
                base = base + WINDOW_SIZE;
                std::cout << "base窗口后移为:" << base << ",开始下一轮发送"
                    << std::endl;
            } else {
                std::unique_lock<std::mutex> lock(mtx);
                if(getAck==-1){base=0;}
                else{base =getAck+1;}
                std::cout << "未收到所有ack,base窗口后移为:" << base << ",重
                    新发送此部分" << std::endl;
                for (int i = base; i < nextseq; ++i) {
                    sendPacket(window[i]);
                    std::cout << "发送数据包.SEQ=" << window[i].seq << ", 校
                        验码=" << window[i].checksum << std::endl;
                }
            }
        }
        //.......
    }
```

2. 接收端

对于接收端（服务器端），接收窗口大小设置为 1，因此每收到一个数据包，都会发送一次 ack。为了处理丢包，设置 buffer 缓冲区，用于存储乱序抵达的包。

具体设计如下：

- 如果收到期待的数据包（seq==expectedseq, 其中 expectedseq=last_ack+1），将数据写入文件，并检查缓冲区中是否存储有后续数据包的内容；如果有，将缓冲区的内容写入文件。依据已经写入文件的数据包序列号来更新 expectedseq, 并发送 ack = expectedSeq - 1。
- 如果收到的数据包序列号大于期待值（seq>expectedseq), 将数据暂存进缓冲区 buffer, 发送最后一次按序抵达的数据包对应的 ack.
- 如果收到的数据包序列号小于期待值（seq<expectedseq），说明收到了重复的数据包，不写入文件，只发送最后一次按序抵达的数据包对应的 ack.

服务器端的接收、发送逻辑

```
void Receiver::receiveFile(const std::string& filename) {
    //.........
    std::unordered_map<int, Datagram> buffer; // 缓存乱序到达的包
    int expectedSeq = 0;
    int lastAckSent = -1;
    while (true) {
        Datagram packet(CLIENT_PORT, ROUTER_PORT);
        if (!receivePacket(packet)) {
            std::cerr << "接收数据包失败\n";
            continue;
        }

        if (packet.flag == 4) { // FIN
            //断开连接过程.....
            break;
        } else if (packet.seq == expectedSeq) {
            // 按序到达的包
            std::cout << "收到数据包, 序列号: " << packet.seq << ",␣校验码"
                << packet.checksum << std::endl;
            file.write(packet.data, packet.dataSize);

            expectedSeq++;

            // 检查缓存中是否有后续的包
            while (buffer.find(expectedSeq) != buffer.end()) {
                const auto& cachedPacket = buffer[expectedSeq];
                file.write(cachedPacket.data, cachedPacket.dataSize);
                buffer.erase(expectedSeq);
                expectedSeq++;
            }

            // 发送新的ACK, 表示此序号前的所有包都已按序接收
            Datagram ackPacket(SERVER_PORT, ROUTER_PORT);
            ackPacket.ack = expectedSeq - 1;
            ackPacket.flag = 0; // 数据包的ACK
            sendPacket(ackPacket);
            lastAckSent = ackPacket.ack;
```

```
            std::cout << "发送ACK,␣ack=" << ackPacket.ack << std::endl;
        } else if (packet.seq > expectedSeq) {
            // 序列号大于期望的，缓存它
            std::cout << "收到乱序数据包，序列号: " << packet.seq << "，期待
                序列号: " << expectedSeq << std::endl;
            buffer[packet.seq] = packet;

            // 发送最后一次按顺序接收到的包的ACK
            Datagram ackPacket(SERVER_PORT, ROUTER_PORT);
            ackPacket.ack = lastAckSent;
            //ackPacket.ack=packet.seq;//修改成，收到啥发送啥
            ackPacket.flag = 0; // 重复的ACK
            sendPacket(ackPacket);
            std::cout << "发送重复ACK,␣ack=" << ackPacket.ack << std::endl;
        } else {
            // 序列号小于期望的，说明是重复的包，重新发送最后一次的ACK
            std::cout << "收到重复数据包，序列号: " << packet.seq << std::
                endl;
            Datagram ackPacket(SERVER_PORT, ROUTER_PORT);
            ackPacket.ack = lastAckSent;
            ackPacket.flag = 0; // 重复的ACK
            sendPacket(ackPacket);
            std::cout << "发送重复ACK,␣ack=" << ackPacket.ack << std::endl;
        }
    }
//.........
}
```

3. 完整实例（结合图示）

为了更好地理解我设计的协议，我绘制了下面的图示来解释文件发送接收过程中的四种情况。

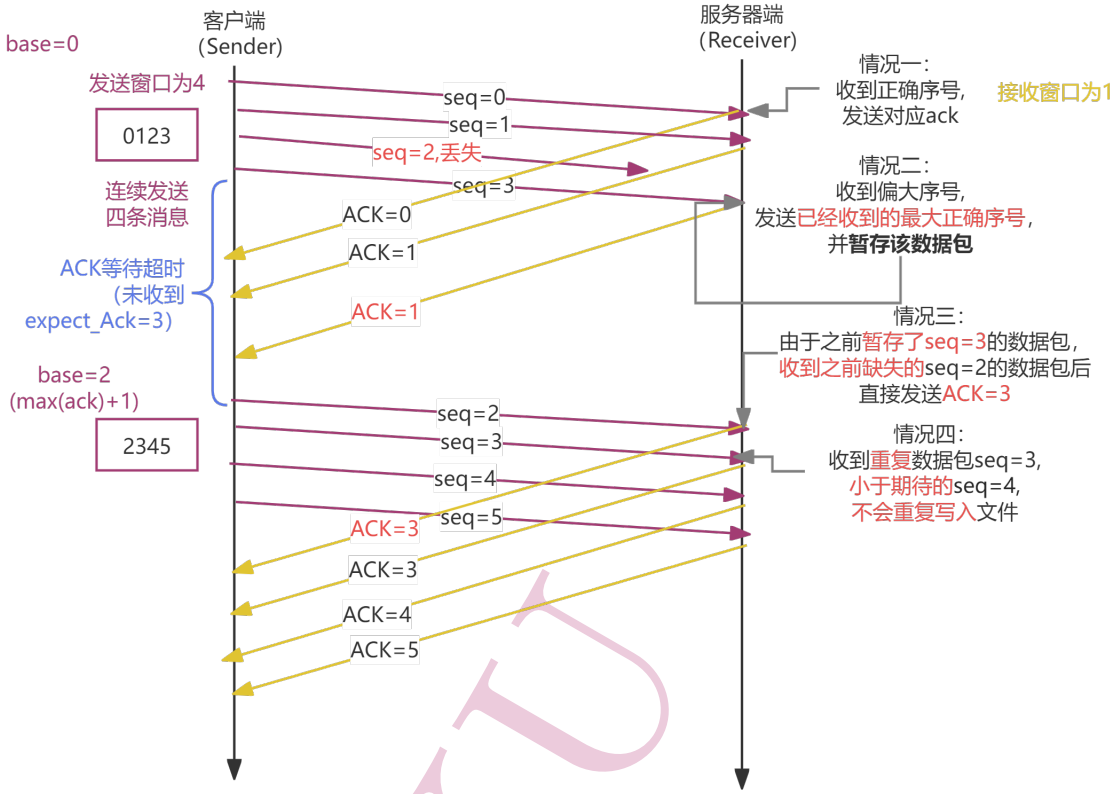


图 7: 如图，展示发送窗口大小为 4 的情况下，客户端与服务器端的消息交互过程。

- 发送窗口大小设置为 4，初始时客户端连续发送序列号为 0,1,2,3 的四个数据包。
- 图中的 seq=0,seq=1 的数据包按序抵达(seq==expectedseq, 其中 expectedseq=last_ack+1), 服务器端会发送对应的 ack=0,ack=1.
- 图中 seq=2 的数据包丢失，于是当 seq=3 的数据包抵达时，服务器端收到了偏大的序号 (seq=3>expectedseq=2)。服务器端会暂存 seq=3 的数据包到缓冲区中，并发送已经收到的最大正确序号（ack=1).
- 客户端一直期待收获到“ack=3”, 等待一段时间没有收到，说明超时。于是基准窗口移动为收到的最大 ack+1, 即 1+1=2，发送序列号为 2，3，4，5 的四个数据包。
- 服务器端收到 seq=2 的数据包后，由于之前暂存了 seq=3 的数据包，因此 ack 更新为 3, 将两个数据包一起写入文件，并发送 ack=3.
- 服务器端再次收到 seq=3 的数据包，不会重复写入，依旧发送 ack=3.
- 服务器端成功接收四个数据包，客户端收到期待的 ack(ack=5), 不再等待定时器到时，立刻更新基准窗口为 6，开始下一轮发送。

二、 运行结果

(一) 建立连接、断开连接

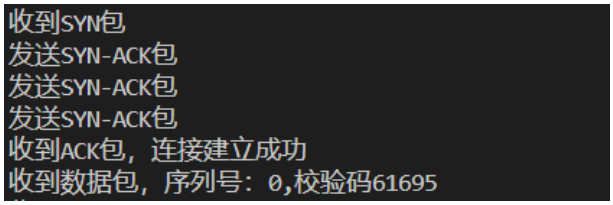


图 8: 服务器端建立连接的输出

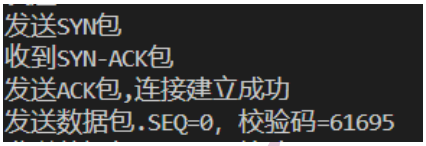


图 9: 客户端建立连接的过程

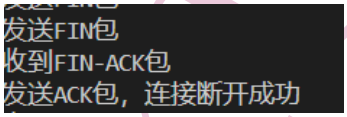


图 10: 客户端断开连接的过程

图 11: 服务器端断开连接的过程

（二） 正常传输（收到正确序列号）

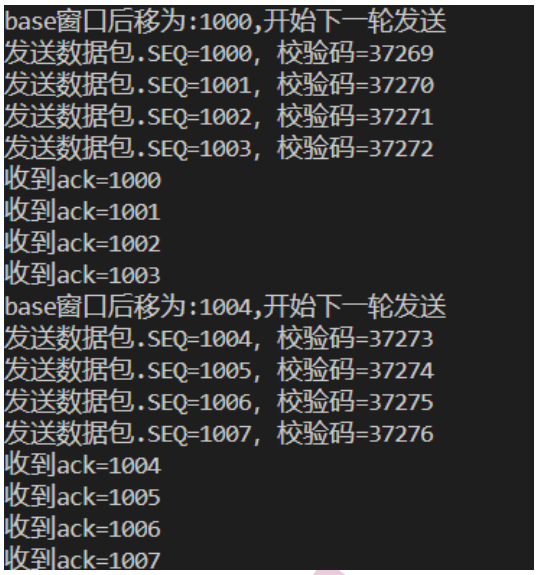


图 12: 客户端发送数据包，收到期待 ack 值，窗口正常后移的情况

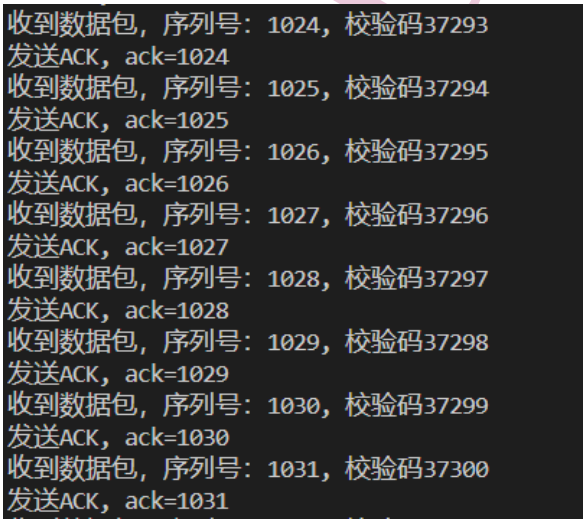


图 13: 服务器端收到期待序列号的数据包，正常发送对应 ack 的情况

(三) 乱序抵达（出现丢包情况，收到偏大/偏小数据包）

```
base窗口后移为:492,开始下一轮发送
发送数据包.SEQ=492, 校验码=36761
发送数据包.SEQ=493, 校验码=36762
发送数据包.SEQ=494, 校验码=36763
发送数据包.SEQ=495, 校验码=36764
收到ack=492
收到ack=492
收到ack=492
未收到所有ack,base窗口后移为:493,重新发送此部分
发送数据包.SEQ=493, 校验码=36762
发送数据包.SEQ=494, 校验码=36763
发送数据包.SEQ=495, 校验码=36764
发送数据包.SEQ=496, 校验码=36765
收到ack=495
收到ack=495
收到ack=495
收到ack=496
```

图 14: 客户端发送时出现丢包现象，丢失 seq=493 的数据包，收到的 ack 值为 492. 定时器超时后，base 窗口后移为 493, 发送从 493 到 496 的数据包。

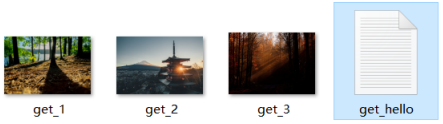
```
收到数据包，序列号：492，校验码36761
发送ACK，ack=492
收到乱序数据包，序列号：494，期待序列号：493
发送重复ACK，ack=492
收到乱序数据包，序列号：495，期待序列号：493
发送重复ACK，ack=492
收到数据包，序列号：493，校验码36762
发送ACK，ack=495
收到重复数据包，序列号：494
发送重复ACK，ack=495
收到重复数据包，序列号：495
发送重复ACK，ack=495
收到数据包，序列号：496，校验码36765
发送ACK，ack=496
```

图 15: 服务器端未收到 seq=493 的数据包, 之后再收到 seq=494,495 的数据包都仅发送 ack=492.

三、 传输结果及分析

(一) 文件传输结果

下表设置的超时重传时间为 10ms.



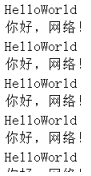


图 16: 传输结果如图，均能正常接收

(二) 正常传输条件下，不同窗口大小的性能对比

文件名	传输时间（s）	吞吐率 (字节/s)	窗口大小	丢包率	延时
helloworld.txt	6.07	272968	4	0	0
1.jpg	7.45	249357	4	0	0
2.jpg	22.97	256876	4	0	0
3.jpg	53.0005	225838	4	0	0

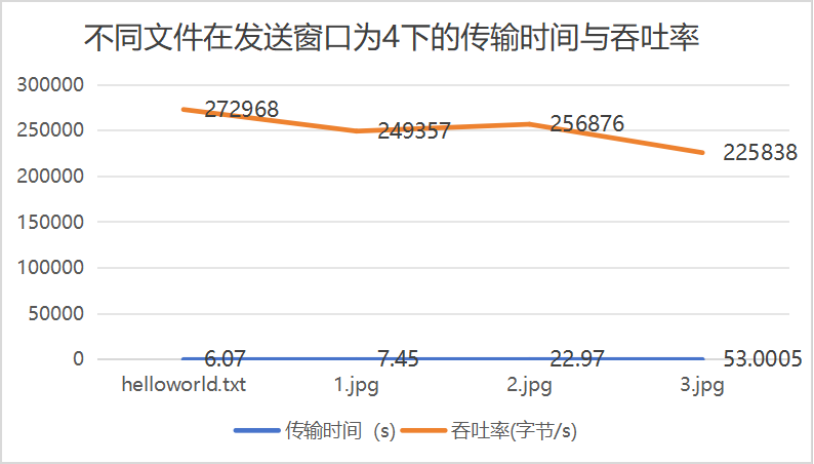


图 17: 不同文件在发送窗口为 4 下的传输时间与吞吐率

从表格、折线图中可以看出，不同文件的吞吐率相差不大，传输时间则有明显差距。传输时间与文件大小接近正比，即文件越大，传输时间越长。

(三) 不同窗口大小性能对比

下表设置的超时重传时间为 10ms.

文件名	传输时间（s）	吞吐率 (字节/s)	窗口大小	丢包率	延时
helloworld.txt	6.07	272968	4	0	0
helloworld.txt	5.03246	329229	8	0	0
helloworld.txt	3.56118	465248	16	0	0
helloworld.txt	2.38237	695456	32	0	0

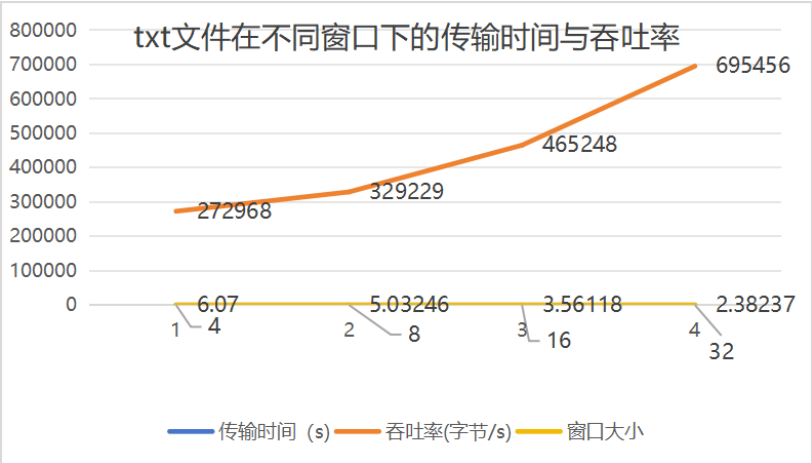


图 18: helloword.txt 文件在不同窗口下的传输时间与吞吐率

从表中，折线图中可以看出，窗口的扩大一定程度上能增大吞吐率，减少传输时间，但吞吐率的增长倍数小于窗口的扩大倍数。而窗口的扩大倍数越大，吞吐率的增长倍数就越接近窗口的扩大倍数。

（四） 不同丢包率的性能对比

为了应对丢包和延时，将超时重传时间（也是一些辅助的等待时间）从 10ms 扩大至 100ms.

文件名	传输时间（s）	吞吐率 (字节/s)	窗口大小	丢包率	延时
helloworld.txt	24.7512	66939.6	8	0	0
helloworld.txt	56.576	29285.1	8	3	0
helloworld.txt	93.5577	17709.2	8	5	0
helloworld.txt	100.282	16521.6	8	7	0

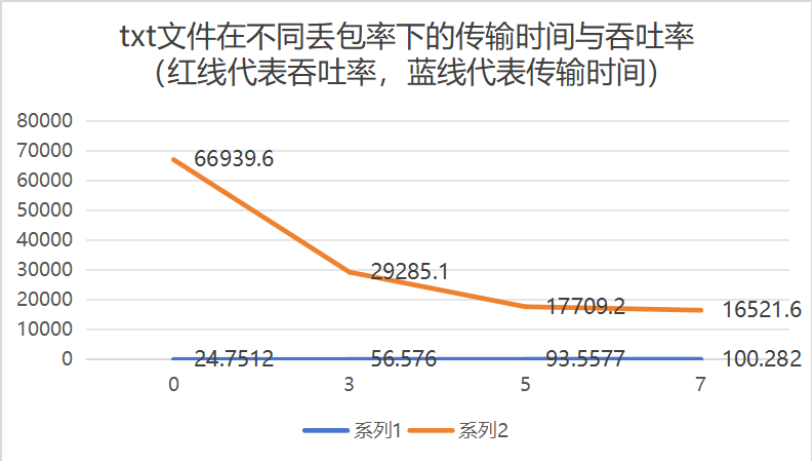


图 19: txt 文件在不同丢包率下的传输时间与吞吐率

如表和图，随着丢包率的增加，吞吐率下降，传输时间增长。但随着丢包率的越来越大，丢包率对吞吐率的影响逐步减小，吞吐率的变化趋于平缓。

（五） 不同延时的性能对比

为了应对丢包和延时，将超时重传时间（也是一些辅助的等待时间）从 10ms 扩大至 100ms.

文件名	传输时间（s）	吞吐率 (字节/s)	窗口大小	丢包率	延时
helloworld.txt	24.7512	66939.6	8	0	0
helloworld.txt	28.4214	58295.1	8	0	1
helloworld.txt	35.4619	46721.4	8	0	3
helloworld.txt	47.8036	34659.1	8	0	5

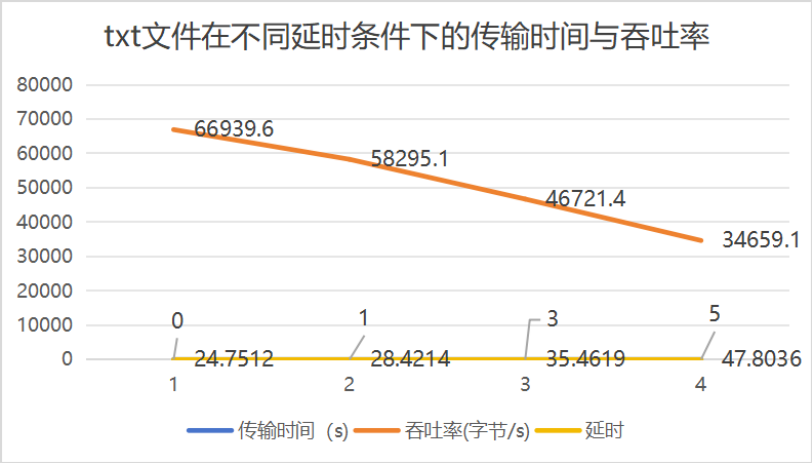


图 20: txt 文件在不同延时条件下的传输时间与吞吐率

如表和图，随着延时的增加，吞吐率逐步降低，传输时间逐步增加。随着延时的线性扩大，吞吐率也线性降低。

四、 实验中遇到的问题与解决方案

（一） 滑动窗口基准值不正确

最开始实现代码后，滑动窗口基准值 base 并未像预期的一样进行正确更新。

仔细研究后发现，base,ackReceived 等值在多个线程中都有访问，需要采用互斥量对其进行保护。修改后能正常运行。

（二） 序列号为 0 的数据包的丢包处理

实验过程中,当序列号为 0 的数据包丢包后,发送端会发送 ack=65535, 接收端收到 ack=65535 后会出现问题。

调试后发现，这是因为 ack 定义为无符号整数，当赋值为-1 时会溢出成 65535.

在服务器端单独处理 ack=65535 的数据包，即可解决。