南開大學
Nankai University

南　　开　　大　　学

网 络 空 间 安 全 学 院

计算机网络课程报告

基于 UDP 服务设计可靠传输协议 3-1

学号：2013018

姓名：许健

年级：2020 级

专业：信息安全

2022 年 11 月 19 日

目录

一、 实验要求 **1**

二、 协议设计 **1**

（一） 协议的语法、语义 1

（二） 协议的时序 2

三、 连接建立与释放 **2**

（一） 建立连接 2

（二） 释放连接 4

四、 可靠机制 **4**

（一） 差错检测 4

（二） 超时重传 5

五、 文件传输 **6**

（一） 上传文件名 6

（二） 上传文件内容 6

六、 结果展示与性能分析 **8**

（一） 结果展示 8

（二） 性能分析 9

七、 实验反思 **11**

八、 后记 **11**

一、 实验要求

1. 利用数据包套接字在用户空间实现面向连接的可靠数据传输，功能包括：建立连接、差错检测、确认重传等。流量控制采用停等机制。
2. 实现单向传输
3. 对于每一个任务要求给出详细的协议设计
4. 完成给定测试文件的传输，显示传输时间和平均吞吐率
5. 性能测试指标：吞吐率、时延，给出图形结果并进行分析
6. 完成详细的实验报告
7. 编写的程序应结构清晰，具有较好的可读性
8. 提交程序源码和实验报告

二、 协议设计

(一) 协议的语法、语义

本次设计的协议头部共包括四个字段：

1. type 字段定义发送消息的类型。
2. seq 字段是发送信息的序列号，由于实现的是单向传输，所以省去了 ack 字段。
3. checksum 校验和字段，校验数据包的包头和数据部分
4. length 标识传输数据的长度

消息类型 type 有如下几种：建立连接（SYN）、回复（ACK）、释放连接（FIN）、发送文件名（FILE）、上传文件（SEND）、是否为最后一个文件块（LAST）。

协议头

```

#pragma once(1)
struct Head {
        u_char type;//状态位
        u_char seq;//序列号,目前0-255
        u_short checksum;//校验和
        u_short length;//数据长度
        //接收窗口、紧急窗口
        //optional字段
};
#pragma once()
#define FILE 0x20
#define SEND 0x10
#define LAST 0x08
#define SYN 0x04
```

```
#define FIN 0x02
#define ACK 0x01
```

(二) 协议的时序

发送文件包括如下步骤:

1. 客户端与服务端三次握手建立连接
2. 客户端发送文件名，服务端回复 ACK
3. 客户端上传文件，服务端接收（停等机制）
4. 客户端与服务端两次挥手释放连接

三、 连接建立与释放

(一) 建立连接

仿照 TCP 采用三次握手机制建立连接，主要协商 seq 起始值。(本次实验默认为 0 开始，也可以选择一个随机数避免冲突）连接建立的过程中加入了差错检测和超时重传机制，客户端和服务端也会校验 type 字段是否正确。

1. 客户端发送 SYN 请求建立连接，seq 为 0
2. 服务端回复 SYN|ACK，ack 为 1
3. 客户端回复 ACK，seq 为 1

建立连接

```
//列出客户端shake_hand函数，服务器端类似
bool ShakeHand() {
        Head h1;
        h1.seq = Seq;
        h1.length = 0;
        h1.type = SYN;
        h1.checksum = 0;
        h1.checksum = CheckSum((u_short*)&h1, sizeof(Head));
        if (sendto(Client, (char*)&h1, sizeof(h1), 0, (sockaddr*)&ServerAddr,
            sizeof(ServerAddr)) == -1) {
                throw("Error: fail to send messages!");
                return false;
        }
        Seq++;
        cout << "Client: " << "[SYN]" << "Seq = " << int(h1.seq) << endl;

        Begin = clock();
        Head recv;
        int len = sizeof(ServerAddr);
```

```
19          while (true) {
20              if (recvfrom(Client, (char*)&recv, sizeof(recv), 0, (SOCKADDR
                    *)&ServerAddr, &len) > 0) {
21                  if (CheckSum((u_short*)&recv, sizeof(recv)) == 0 &&
                        recv.type == (SYN | ACK) && recv.seq == Seq) {
22                      //验证消息类型、校验和、序列号
23                      cout << "Server [SYN, ACK]" << "ack = " <<
                            int(Seq) << endl;
24                      break;
25                  }
26                  else {
27                      //差错重传
28                      if (sendto(Client, (char*)&h1, sizeof(h1), 0,
                            (sockaddr*)&ServerAddr, sizeof(
                            ServerAddr)) == -1) {
29                          throw("Error: fail to send messages!"
                                );
30                          return false;
31                      }
32                      Begin = clock();
33                  }
34              }
35              else if (clock() - Begin > MAX_WAIT_TIME) {
36                  //超时重传
37                  if (sendto(Client, (char*)&h1, sizeof(h1), 0, (
                        sockaddr*)&ServerAddr, sizeof(ServerAddr)) == -1)
                         {
38                      return false;
39                  }
40                  Begin = clock();
41              }
42          }
43
44          Head h3;
45          h3.seq = Seq;
46          h3.type = ACK;
47          h3.length = 0;
48          h3.checksum = 0;
49          h3.checksum = CheckSum((u_short*)&h3, sizeof(h3));
50          if (sendto(Client, (char*)&h3, sizeof(h3), 0, (sockaddr*)&ServerAddr,
                sizeof(ServerAddr)) == -1) {
51              return false;
52          }
53          Seq++;
54          cout << "client: " << "[ACK]" << "seq=" << int(h3.seq) << endl;
55          return true;
56      }
```

(二) 释放连接

由于本次实验实现的是单向文件传送，因此只有两次挥手，同样确保了可靠机制。

1. 客户端发送 FIN|ACK 请求释放连接，seq 为 u
2. 服务端回复 ACK，ack 为 u+1

建立连接

```
//列出客户端wave_hand函数，服务器端类似
bool WaveHand() {
        Head recv;
        int len = sizeof(ClientAddr);
        while (true) {
                if (recvfrom(Server, (char*)&recv, sizeof(recv), 0, (sockaddr
                    *)&ClientAddr, &len) > 0) {
                        if (CheckSum((u_short*)&recv, sizeof(recv)) == 0 && (
                            recv.type == (FIN | ACK))) {
                                cout << "Client [FIN, ACK]" << "Seq = " <<
                                    int(recv.seq) << endl;
                                Seq = recv.seq + 1;
                                break;
                        }
                }
        }
        Head h2;
        h2.seq = Seq;
        h2.type = ACK;
        h2.checksum = 0;
        h2.checksum = CheckSum((u_short*)&h2, sizeof(h2));
        if (sendto(Server, (char*)&h2, sizeof(h2), 0, (sockaddr*)&ClientAddr,
            sizeof(ClientAddr)) == -1) {
                throw("Error: fail to send messages!");
                return false;
        }
        cout << "Server [ACK]" << "ack = " << int(Seq) << endl;
        Seq++;
        return true;
}
```

四、 可靠机制

(一) 差错检测

差错检测主要校验和字段，如果接收到的 checksum 为 0，则数据正确，否则包损坏。校验和计算按每 16 位求和并在高位补 0 得到得出一个 32 位的求和结果；如果这个 32 位的求和结果，高 16 位不为 0，则高 16 位加低 16 位并且高位补 0 再得到一个 32 位的结果。

计算校验和

```
u_short CheckSum(u_short* buf, int length) {
        int count = (length + 1) / 2;
        u_int sum = 0;
        while (count--) {
                sum += *buf++;
                if (sum & 0xffff0000) {
                        sum &= 0xffff;
                        sum++;
                }
        }
        return ~(sum & 0xffff);
}
```

(二) 超时重传

超时重传机制是为了防止数据包丢失导致的程序异常。需要将 recvfrom 函数设置为非阻塞，增添一个计时器，如果超时则重传数据包，重新计时（数据包出现差错也需要计时）

差错重传与超时重传

```
while (true) {
    if (recvfrom(Client, (char*)&recv, sizeof(recv), 0, (SOCKADDR*)&
        ServerAddr, &len) > 0) {
        if (CheckSum((u_short*)&recv, sizeof(recv)) == 0 && recv.type == (SYN
            | ACK) && recv.seq == Seq) {
            cout << "Server [SYN, ACK]" << "ack = " << int(Seq) << endl;
            break;
        }
        else {//差错重传
            if (sendto(Client, (char*)&h1, sizeof(h1), 0, (sockaddr*)&
                ServerAddr, sizeof(ServerAddr)) == -1) {
                throw("Error: fail to send messages!");
                return false;
            }
            Begin = clock();
        }
    }
    else if (clock() - Begin > MAX_WAIT_TIME) {//超时重传
        if (sendto(Client, (char*)&h1, sizeof(h1), 0, (sockaddr*)&ServerAddr,
            sizeof(ServerAddr)) == -1) {
            throw("Error: fail to send messages!");
            return false;
        }
        Begin = clock();
    }
}
```

五、 文件传输

(一) 上传文件名

上传文件之前客户端需要告诉服务端存储文件的名字,通过 $SendFileName(stringFileName)$ 函数完成，类似握手挥手，该报文同样要占用一个序列号。

SendFileName(string FileName)

```
void SendFileName(string FileName) {
        cout << "传送文件名ing..." << endl;
        Head head;
        head.seq = Seq;
        head.length = FileName.length();
        head.type = FILE;
        head.checksum = 0;
        memset(sendBuf, 0, sizeof(sendBuf));
        memcpy(sendBuf, &head, sizeof(head));
        memcpy(sendBuf + sizeof(head), FileName.c_str(), head.length);
        head.checksum = CheckSum((u_short*)sendBuf, sizeof(head) + head.
            length);
        memcpy(sendBuf, &head, sizeof(head));
        if (sendto(Client, sendBuf, head.length + sizeof(head), 0, (SOCKADDR
            *)&ServerAddr, sizeof(SOCKADDR)) == -1) {
                cout << "Error: fail to send messages!" << endl;
        }
        cout << "Client: " << "[FILE]" << "seq=" << int(Seq) << endl;
        Seq++;
        Begin = clock();
        memset(recvBuf, 0, sizeof(recvBuf));
        int len = sizeof(ServerAddr);
        while (true) {
            //接受服务端的ack...
    }
```

(二) 上传文件内容

$Send_Message(char*data, intlength)$ 函数完成文件内容的上传，传递的参数 data 是文件数据缓冲区，length 是文件大小。SendTime 统计发送文件的时间以计算吞吐率。对于发送的文件块，type 为 SEND，如果是最后一个文件块作要将 LAST 位置 1，作为传送结束的标记。文件传送过程中也实现了可靠机制，在结果展示与性能分析一节会演示。

$Send_Message()$

```
SendTime = clock_t();
TotalNum = length / MAX_SEND_SIZE + (length % MAX_SEND_SIZE != 0);
cout << "传送数据ing..." << endl;
SendNum = 0;
Head head;
while (true) {
```

```
7       if (SendNum == TotalNum) {
8           break;
9       }
10      head.seq = Seq;
11      head.length = MAX_SEND_SIZE;
12      head.type = SEND;
13      head.checksum = 0;
14      if (SendNum == TotalNum - 1) {
15          head.type = SEND | LAST;
16          head.length = length % MAX_SEND_SIZE;
17      }
18      memset(sendBuf, 0, sizeof(sendBuf));
19      memcpy(sendBuf, &head, sizeof(head));
20      memcpy(sendBuf + sizeof(head), data + SendNum * MAX_SEND_SIZE, head.
            length);
21      head.checksum = CheckSum((u_short*)sendBuf, sizeof(head) + head.length);
22      memcpy(sendBuf, &head, sizeof(head));
23      if (sendto(Client, sendBuf, head.length + sizeof(head), 0, (SOCKADDR*)&
            ServerAddr, sizeof(SOCKADDR)) == -1) {
24          cout << "Error: fail to send messages!" << endl;
25      }
26      cout << "Client: " << "[Send]" << "seq=" << int(Seq) << "   length:" <<
            head.length << "    checksum:" << head.checksum << endl;;
27      SendNum++;
28      Seq++;
29      Begin = clock();
30      int len = sizeof(ServerAddr);
31      while (true) {
32          memset(recvBuf, 0, sizeof(recvBuf));
33          if (recvfrom(Client, (char*)&recvBuf, sizeof(recvBuf), 0, (SOCKADDR*)
                &ServerAddr, &len) > 0) {
34              Head recv;
35              memcpy((char*)&recv, recvBuf, sizeof(recv));
36              if (CheckSum((u_short*)&recvBuf, sizeof(recvBuf)) == 0 && recv.
                    type == ACK && recv.seq == Seq) {
37                  //验证消息类型、校验和、序列号
38                  break;
39              }
40              else {
41                  //差错重传
42                  if (sendto(Client, sendBuf, head.length + sizeof(head), 0, (
                        SOCKADDR*)&ServerAddr, sizeof(SOCKADDR)) == -1) {
43                      throw("Error: fail to send messages!");
44                  }
45                  cout << "出错重传序列号为" << int(Seq) - 1 << "报文" << endl;
46                  Begin = clock();
47              }
48          }
```

```
        else if (clock() - Begin > MAX_WAIT_TIME) {
            //超时重传
            if (sendto(Client, sendBuf, head.length + sizeof(head), 0, (
                SOCKADDR*)&ServerAddr, sizeof(SOCKADDR)) == -1) {
                throw("Error: fail to send messages!");
            }
            cout << "超时重传序列号为" << int(Seq) - 1 << "报文" << endl;
            Begin = clock();
        }
    }
}
SendTime = clock() - SendTime;
```

六、 结果展示与性能分析

(一) 结果展示

客户端和服务端打印的日志，包括 SOCTET 初始化、三次握手、上传文件名、上传文件等过程，客户端会打印序列号 seq 和每次发送数据的长度 length，服务端会打印接受序列号 ack。(不考虑丢包和包出错)

```
WSADATA init success!
create socket of client success!
connecting...
Client: [SYN]Seq = 0
Server [SYN, ACK]ack = 1
client: [ACK]seq=1
Handshake succeeded!
选择要上传的文件:
[1]1.jpg
[2]2.jpg
[3]3.jpg
[4]helloworld.txt
1
传送文件名ing...
Client: [FILE]seq=2
传送数据ing...
Client: [Send]seq=3     length:10000    checksum:58900
Client: [Send]seq=4     length:10000    checksum:37022
Client: [Send]seq=5     length:10000    checksum:42376
Client: [Send]seq=6     length:10000    checksum:32296
Client: [Send]seq=7     length:10000    checksum:12915
Client: [Send]seq=8     length:10000    checksum:47930
Client: [Send]seq=9     length:10000    checksum:25649
Client: [Send]seq=10    length:10000    checksum:11075
Client: [Send]seq=11    length:10000    checksum:1975
Client: [Send]seq=12    length:10000    checksum:30327
```

图 1: 客户端日志

图 2: 服务端日志

可以看到发送的结果，共发送 11968994 字节，发送时间为 6.963s，吞吐率为 1718KB/s。文件上传后，客户端与服务端两次挥手释放连接。

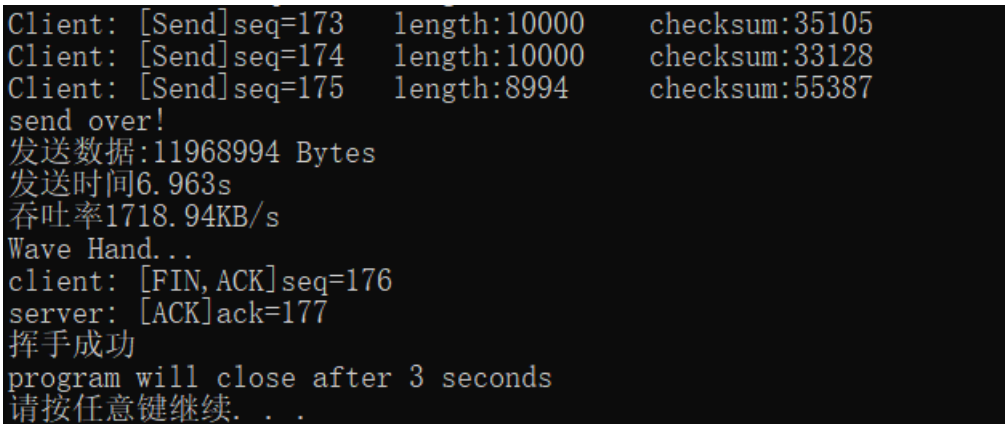

图 3: 结果展示

(二) 性能分析

上一节结果展示中，无丢包情况下性能可以达到 1718KB/s，而实际物理网络中不会这么理想。我们人为编写一个丢包函数，设置丢包率为 0.01，并设置包可能出错的情况。实验结果如图4和图5所示，可以看到吞吐率下降到 311KB/s，传递速度很慢。这也是停止等待协议的缺陷，受超时重传的影响较大。后续的实验我们将采用基于滑动窗口的 GBN/SR 算法实现流水线传输，对比性能的变化。

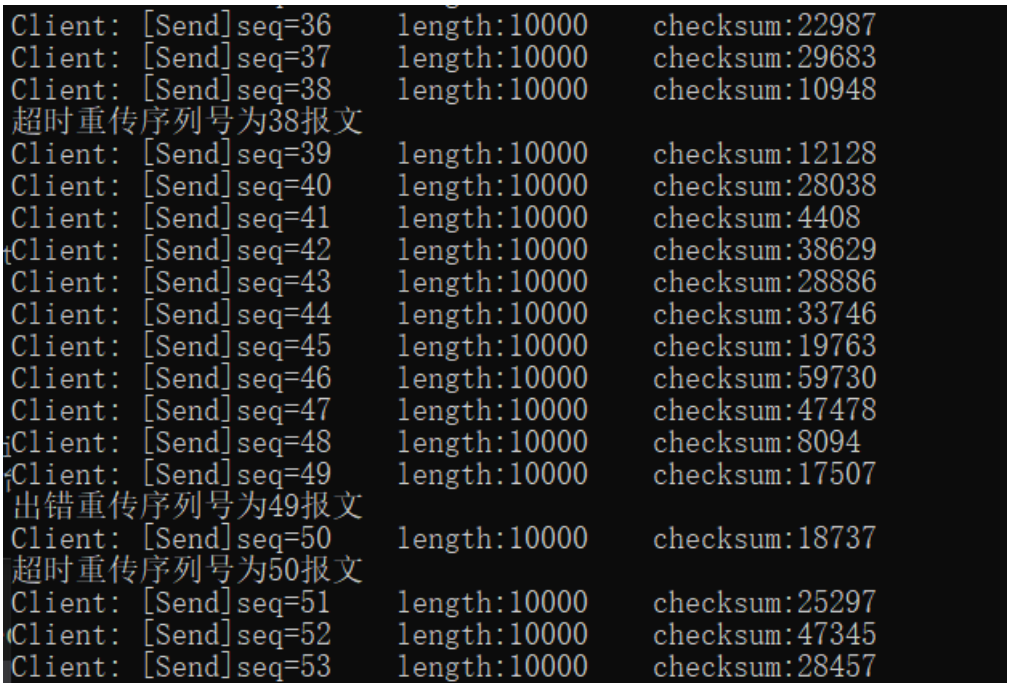

图 4: 丢包与出错重传

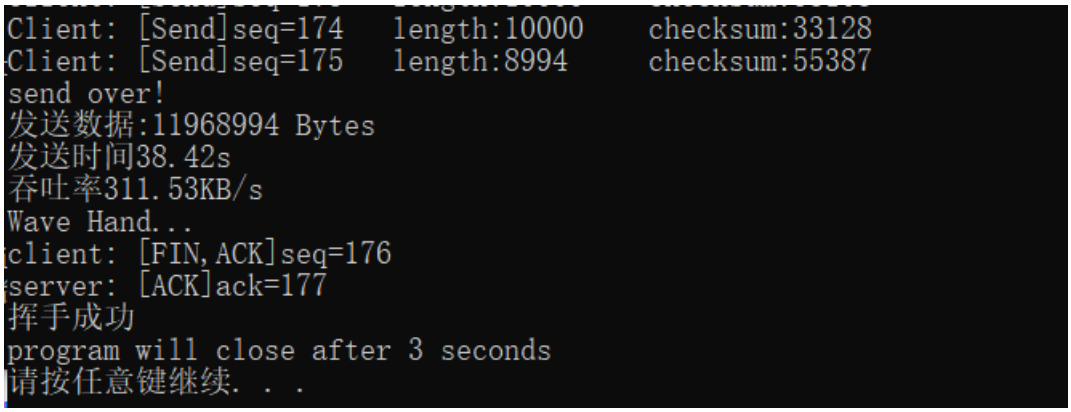

图 5: 传输时间与吞吐率

人为设置丢包与包出错

```
BOOL lossInLossRatio(float lossRatio) {
        int lossBound = (int)(lossRatio * 100);
        int r = rand() % 101;
        if (r <= lossBound) {
                return TRUE;
        }
        return FALSE;
}
if (Seq == 50) {
        reply.type = -1;
}
```

七、 实验反思

基于 UDP 协议实现可靠传输，可以把 TCP 可靠传输的特性（序列号、确认应答、超时重传、流量控制、拥塞控制）在应用层实现一遍。但是 TCP 天然支持可靠传输，为什么还需要基于 UDP 实现可靠传输呢？这不是重复造轮子吗？

所以，我们要先弄清楚 TCP 协议有哪些痛点？而这些痛点是否可以在基于 UDP 协议实现的可靠传输协议中得到改进？TCP 协议四个方面的缺陷：

1. 升级 TCP 的工作很困难
2. TCP 建立连接的延迟
3. TCP 存在队头阻塞问题
4. 网络迁移需要重新建立 TCP 连接

现在市面上已经有基于 UDP 协议实现的可靠传输协议的成熟方案了，那就是 QUIC 协议，已经应用在了 HTTP/3。

QUIC 实现了两种级别的流量控制，分别为 Stream 和 Connection 两种级别。QUIC 协议中同一个 Stream 内，滑动窗口的移动仅取决于接收到的最大字节偏移（尽管期间可能有部分数据未被接收），而对于 TCP 而言，窗口滑动必须保证此前的 packet 都有序的接收到了，其中一个 packet 丢失就会导致窗口等待。QUIC 对于队头阻塞问题解决得更加彻底。

QUIC 协议当前默认使用了 TCP 的 Cubic 拥塞控制算法（我们熟知的慢开始、拥塞避免、快重传、快恢复策略），同时也支持 Reno、BBR、PCC 等拥塞控制算法。QUIC 是处于应用层的，应用程序层面就能实现不同的拥塞控制算法，不需要操作系统，不需要内核支持。这是一个飞跃，因为传统的 TCP 拥塞控制，必须要端到端的网络协议栈支持，才能实现控制效果。而内核和操作系统的部署成本非常高，升级周期很长，所以 TCP 拥塞控制算法迭代速度是很慢的。而 QUIC 可以随浏览器更新，QUIC 的拥塞控制算法就可以有较快的迭代速度。TCP 更改拥塞控制算法是对系统中所有应用都生效，无法根据不同应用设定不同的拥塞控制策略。但是因为 QUIC 处于应用层，所以就可以针对不同的应用设置不同的拥塞控制算法，这样灵活性就很高了。

QUIC 协议没有用四元组的方式来“绑定”连接，而是通过连接 ID 来标记通信的两个端点，客户端和服务器可以各自选择一组 ID 来标记自己，因此即使移动设备的网络变化后，导致 IP 地址变化了，只要仍保有上下文信息（比如连接 ID、TLS 密钥等），就可以“无缝”地复用原连接，消除重连的成本，没有丝毫卡顿感，达到了连接迁移的功能。

因此后续的实验如果有机会的话，应该尽可能实现类似 QUIC 这样的可靠 UDP 传输协议，而不是对 TCP 照搬照抄。

八、 后记

做一个善于思考的人。本次实验 github 链接