南開大學
Nankai University

计算机和网络空间安全学院

机器学习实验报告

大作业 3-1：基于 UDP 服务的可靠传输协议

姓名：魏伯繁

学号：2011395

专业：信息安全

2022 年 11 月 19 日

目录

1 实验要求 **2**

2 实验设计 **2**

2.1 实验设计说明 2

2.2 数据报格式 2

2.3 常数设计 4

3 流程说明 **4**

3.1 三次握手 4

3.2 四次挥手 6

3.3 传输过程 7

3.4 接收过程 8

4 结果展示 **10**

5 辅助函数 **10**

5.1 校验和 11

5.2 传输结束通知 11

6 总结 **12**

1 实验要求

利用数据报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，完成给定测试文件的传输
数据报套接字：UDP；
建立连接：实现类似 TCP 的握手、挥手功能；
差错检测：计算校验和；
确认重传：rdt2.0、rdt2.1、rdt2.2、rdt3.0 等，亦可自行设计协议；
单向传输：发送端、接收端；
有必要日志输出。

2 实验设计

2.1 实验设计说明

本次实验是计算机网络课程大作业的第一次实验，旨在完善基本的连接建立、销毁流程的包含差错处理的实现以及实现最基本的传输协议–停等机制的设计。并在此基础上完成后续大作业代码的实现。

在本次实验中，我以 rdt3.0 为蓝本，根据实验具体的实验传输要求设计实现自己的传输协议

2.2 数据报格式

首先来介绍在本次实验中使用的数据报格式，下图为本次使用的数据报格式的示意图

0-15	16-31	32-47	48-64
checksum	seq	ack	flag
length	source_ip	des_ip	source_port
desport	data	data	....

图 2.1: 数据报格式示意图

checksum 表示校验和，用于进行差错检测
seq 用来表示本次传输的数据报的编号
ack 表示下一次希望收到的 seq 编号
flag 是标志位，通常用于表示一些具有特殊含义的数据报
length 表示了 data 段的长度（最大为 256）
source_ip 表示源 ip 地址
des_ip 表示目的 ip 地址
source_port 表示源端口号
des_port 表示目的端口号
data 表示传输的数据，其长度由 data 段的数据规定
其具体的代码实现如下：

```

//数据头
struct Header {
    u_short checksum; //16 位校验和
    u_short seq; //8 位序列号，因为是停等，所以只有最低位实际上只有 0 和 1 两种状态
    u_short ack; //8 位 ack 号，因为是停等，所以只有最低位实际上只有 0 和 1 两种状态
    u_short flag;//8 位状态位 倒数第一位 SYN，倒数第二位 ACK，倒数第三位 FIN，倒数第四位是结束位
    u_short length;//8 位长度位
    u_short source_ip; //16 位 ip 地址
    u_short des_ip; //16 位 ip 地址
    u_short source_port; //16 位源端口号
    u_short des_port; //16 位目的端口号
    Header() {//构造函数
        checksum = 0;
        source_ip = SOURCEIP;
        des_ip = DESIP;
        source_port = SOURCEPORT;
        des_port = DESPORT;
        seq = 0;
        ack = 0;
        flag = 0;
        length = 0;
    }
};

```

关于 flag 段：我们有如下定义：其中
SYN 作为初始化挥手的请求标志
ACK 作为确认标志
SYN_ACK 作为第二次挥手时应该具备的 flag 段值
OVER 表示传输结束时告知服务端的数据报标识位
OVER_ACK 表示服务端收到并确认传输结束数据报
FIN 用作表示尝试断开即挥手时的标识位
FIN_ACK 表示收到并确认尝试断开请求数据报
FINAL_CHECK 表示服务端接受到客户端的第四次挥手，避免丢包死锁

```

const unsigned char SYN = 0x1;//OVER=0,FIN=0,ACK=0,SYN=1
const unsigned char ACK = 0x2;//OVER=0,FIN=0,ACK=1,SYN=0
const unsigned char SYN_ACK = 0x3;//OVER=0,FIN=0,ACK=1,SYN=1
const unsigned char OVER = 0x8;//OVER=1,FIN=0,ACK=0,SYN=0

```

```
const unsigned char OVER_ACK = 0xA;//OVER=1,FIN=0,ACK=1,SYN=0
const unsigned char FIN = 0x10;//FIN=1,OVER=0,FIN=0,ACK=0,SYN=0
const unsigned char FIN_ACK = 0x12;//FIN=1,OVER=0,FIN=0,ACK=1,SYN=0
const unsigned char FINAL_CHECK=0x20;//FC=1.FIN=0,OVER=0,FIN=0,ACK=0,SYN=0

```

2.3 常数设计

除此之外，在数据报传输的过程中也需要设计很多常量来保证程序的稳步执行，在本程序中制定如下常量：

blockmode 和 unblockmode 起到了相反的作用，他们是用来设置 recvfrom 函数的阻塞与否的，当我们需要在 recvfrom 的返回值非正的时候做判断的时候就需要运用到这两个常数

MAX_DATA_LENGTH 的作用是用于规定能发送的最大的数据报长度（不包括首部）

接下来四个分别是源 ip 地址、目的 ip 地址、源端口号和目的端口号

最后一个是超时重传的最长时间，设置为 0.2s

```

u_long blockmode = 0;
u_long unblockmode = 1;
const unsigned char MAX_DATA_LENGTH = 0xff;
const u_short SOURCEIP = 0x7f01;
const u_short DESIP = 0x7f01;
const u_short SOURCEPORT = 8888;//源端口是 8888
const u_short DESPORT = 8887;//客户端端口号是 8887
const double MAX_TIME = 0.2*CLOCKS_PER_SEC;

```

3 流程说明

流程说明中主要包含三个方面：对三次握手的改进、对三次挥手的改进以及对传输、接收数据的流程说明

3.1 三次握手

三次握手的具体实现流程与在理论课上学习的大体一致。具体流程如下图所示：第一次由客户端向服务端发送请求，发送 SYN，服务端确认数据报无误后返回 SYN_ACK 表示确认握手请求，最后再由客户端发送 ACK 表示三次握手结束。

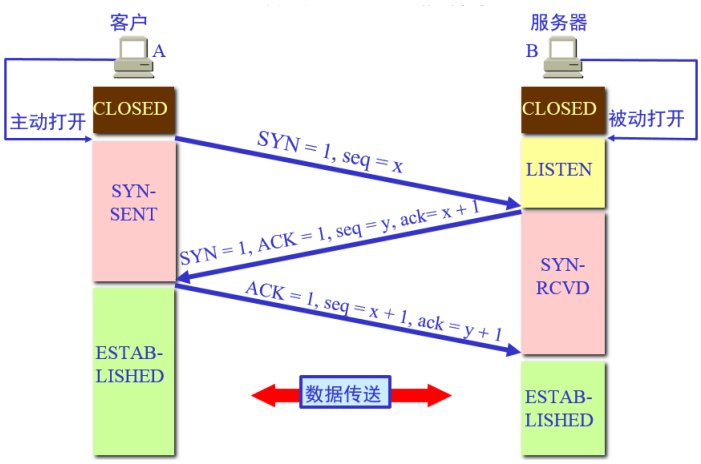

图 3.2: 改进后的三次握手示意图

在本实验中对三次握手的改进主要体现在三次握手后的处理，即第三次握手的数据报发送后该数据报可能会丢包，这将导致服务端无法正常开始，于是本协议规定，当客户端发送第三次握手后需要服务器返回一个再确认数据报，如果客户端在 MAX_TIME 的时限内未收到该数据报则会重传第三次握手消息，在不超过 LINKTIME 的情况下将直到接收到争取的数据报为止。

LINKTIME 是仿照 tcp-ip 协议设计的建立连接计时器，即当建立连接的时间超过所设定的时间时，连接会自动断开。

改进后关于最后一次再确认的服务端部分代码如下：

```
1
2   memcpy(&header, recvshbuffer, sizeof(header));
3       if (header.flag == ACK && vericksum((u_short*)(&header), sizeof(header)) == 0) {
4           cout << "[3] 成功接收第三次握手消息！可以开始接收数据..." << endl;
5           header.source_port = SOURCEPORT;
6           header.des_port = DESPORT;
7           header.flag = ACK;
8           header.source_port = SOURCEPORT;
9           header.des_port = DESPORT;
10          header.ack = (header.seq + 1) % 2;
11          header.seq = 0;
12          header.length = 0;
13          header.checksum = calcksum((u_short*)(&header), sizeof(header));
14          memcpy(sendshbuffer, &header, sizeof(header));
15          sendto(server, sendshbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen);
16          cout << "[EVERYTHING_DONE] 确认信息传输成功...." << endl;
17      }
18      else {
19          cout << "[failed] 不是期待的数据包，正在重传并等待客户端等待重传" << endl;
20          if (clock() - linkClock > 75 * CLOCKS_PER_SEC) {
21              cout << "[failed] 连接超时，服务器自动断开" << endl;
```

```
22                 return -1;
23             }
24             goto SECONDSHAKE;
25         }
26         cout << "[WAITING] 正在等待接收数据...." << endl;
27         return 1;
28
```

3.2 四次挥手

四次挥手的大体流程也和理论课上讲授的一致，先由客户端发起请求，发送 FIN，随后客户端先后发送 ACK 以及 FIN_ACK 来作为确认请求发送给客户端，最后客户端在发送 ACK 后等待两个 MSL 后中断连接。

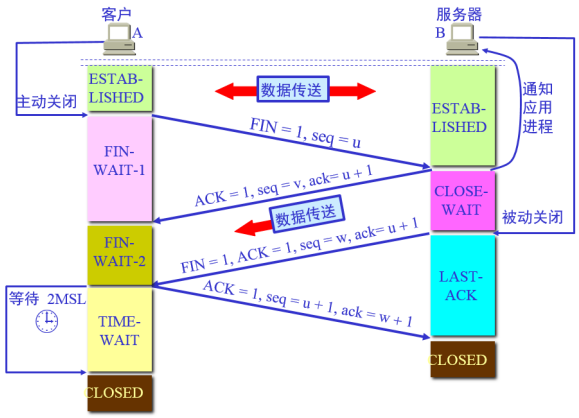

图 3.3: 改进后的四次挥手

关于四次挥手的改机也发生在第四次挥手，如果说第四次挥手的包丢失，服务端将无法正常关闭，所以我们要求服务端在收到第四次挥手🄵消息后发送一个确认数据报，并等待 10 个 MAX_TIME 后再关闭连接，因为如果服务端的包丢失了，那么最多两个 MAX_TIME，客户端就会重发第四次挥手的消息，此时服务端就可以发现自己的再确认消息丢包，于是就会再次发送，这样双方都可以完成连接的正常关闭。

改进后的四次挥手🄵服务端代码改进如下

```
1
2   SEND5:
3       memcpy(&header, recvbuffer, sizeof(header));
4       if (header.flag == ACK && vericksum((u_short*)&header, sizeof(header)) == 0) {
5           header.seq = 0;
6           header.flag = FINAL_CHECK;
7           header.checksum = calcksum((u_short*)&header, sizeof(header));
8           memcpy(sendbuffer, &header, sizeof(header));
9           sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen);
10          cout << " 成功发送确认报文" << endl;
```

```
        }
        start = clock();
        while (recvfrom(server, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr, &rlen) <= 
            if (clock() - start > 10 * MAX_TIME) {
                cout << " 四次挥手结束，已经断开连接" << endl;
                return 1;
            }
        }
        goto SEND5;
    }

    }
```

3.3 传输过程

根据实验要求，本次实验采用停等机制，所以只需要有两个序列号 0、1 就可以满足实验要求，根据停等机制的特性，不支持流水线作业，所以当未收到准确无误的对 0 号序列号确认的 ACK 前不能发送 1 号数据报

根据停等机制这样一种简单的特性，我们可以使用 while 循环来完成对停等机制的实现，因为我实现的是 rdt3.0，所以在传输过程中还要对可能发生的错误进行检测并完成差错重传。网络中可能存在的传输错误大体有两种：丢包和数据报损毁，针对这两种情况，我们可以使用超时重传机制来判定网络中的丢包现象并重新发包，使用计算校验和的方式来判断包的数据完整性。并且使用有限状态机来描述传输过程中具体可能出现的情况。

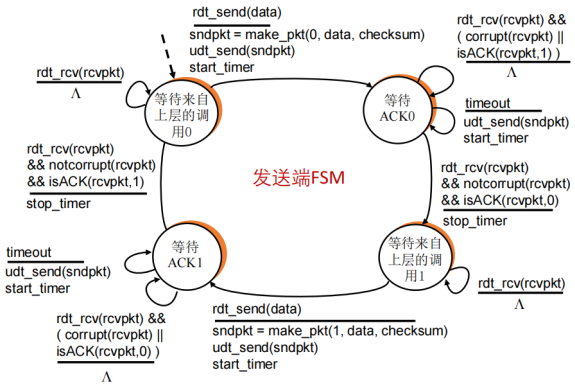

图 3.4: 发送端有限状态机

当发包完成后即进入等待确认状态，并且设置时钟，如果时钟超时还未收到服务端的确认则触发超时重传机制，将重新发送对应的数据报，如果收到了服务端的确认信息，还要检测该确认信息的 ack 是否是发送的（seq+1）%2 并检查数据报是否损毁，如果没有损毁则认为成功接受到确认，并准备发送下一个数据报。具体截取了一部分代码如下图所示：

```

    SEQ1SEND:
```

```
3           //发送 seq=1 的数据包
4           cout << "[1] 准备发送" << "1" << " 号数据包，该数据包大小为:" << ml << " ";
5           cout << " 发送前检验：整体校验为" << vericksum((u_short*)sendbuffer, sizeof(header) + M
6           if (sendto(client, sendbuffer, (sizeof(header) + MAX_DATA_LENGTH), 0, (sockaddr*)&rout
7               cout << "[failed]seq1 数据包发送失败.... 请检查原因" << endl;
8               return -1;
9           }
10          start = clock();
11      SEQ1RECV:
12          //如果收到数据了就不发了，否则延时重传
13          while (recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr, &rlen)
14              if (clock() - start > MAX_TIME) {
15                  if (sendto(client, sendbuffer, (sizeof(header)+MAX_DATA_LENGTH), 0, (sockaddr*
16                      cout << "[failed]seq1 数据包发送失败.... 请检查原因" << endl;
17                      return -1;
18                  }
19                  start = clock();
20                  cout << "[ERROR]seq1 数据包反馈超时.... 正在重传" << endl;
21                  //goto SEQ1SEND;
22              }
23          }
24          //检查 ack 位是否正确，如果正确则准备发下一个数据包
25          memcpy(&header, recvbuffer, sizeof(header));
26          cout << "[GETACK] 接受到的 ack 为: " << header.ack << " 接受到的校验和为: " << vericksu
27          if (header.ack == 0 && vericksum((u_short*)&header, sizeof(header)) == 0) {
28              cout << "[1CHECKED]seq1 的数据包成功接受服务端 ACK，准备发出下一个数据包" << endl;
29          }
30          else {
31              cout << "[ERROR] 服务端未反馈正确的数据包... 正在等待重传..." << endl;
32              goto SEQ1SEND;
33          }
34      }
35
```

3.4 接收过程

接收过程的流程与发送时大同小异，接收端接收发送端发送的数据报并检测数据报的正确性，如果数据报正确则保存如缓冲区，反之则直接丢弃。如果收到正确的数据报应给回 ACK 信号，ack 的值应为（seq+1）%2, 同样在接收端也设置超时重传和差错检验，如果超过固定的时间没有收到发送端的消息则重传 ACK 数据报，同理如果数据报出错则不给回 ACK，等待发送端重发。

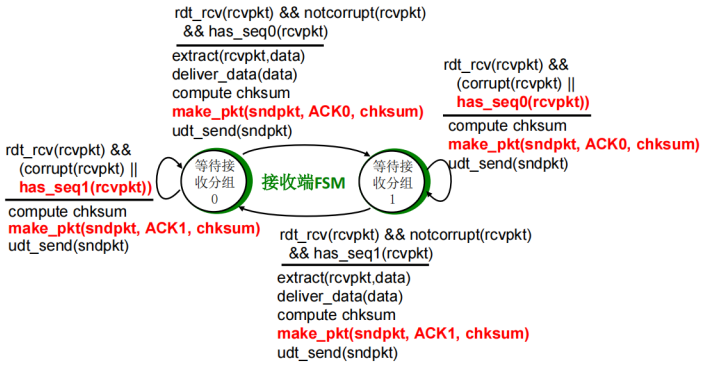

图 3.5: 接收端有限状态机

下面是接收端的部分代码:

```
1
2           if (header.flag == OVER) {
3               //传输结束，等待添加....
4               if (vericksum((u_short*)&header, sizeof(header)) == 0) { if (endreceive()) { retur
5               else { cout << "[ERROR] 数据包出错，正在等待重传" << endl; goto WAITSEQ0; }
6           }
7           cout << header.seq << " " << vericksum((u_short*)recvbuffer, sizeof(header) + MAX_DATA
8           //printheader(header);
9           //printcharstar(recvbuffer, sizeof(header) + MAX_DATA_LENGTH);
10          if (header.seq == 0 && vericksum((u_short*)recvbuffer, sizeof(header)+MAX_DATA_LENGTH)
11              cout << "[OCHECKED] 成功接收 seq=0 数据包" << endl;
12              memcpy(message + messagepointer, recvbuffer + sizeof(header), header.length);
13              messagepointer += header.length;
14              break;
15          }
16          else {
17              cout << "[ERROR] 数据包错误，正在等待对方重新发送" << endl;
18          }
19      }
20      header.ack = 1;
21      header.seq = 0;
22      header.checksum = calcksum((u_short*)&header, sizeof(header));
23      memcpy(sendbuffer, &header, sizeof(header));
24  SENDACK1:
25      if (sendto(server, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen) == -1) {
26          cout << "[failed]ack1 发送失败...." << endl;
27          return -1;
28      }
```

```
29      clock_t start = clock();
30
```

4 结果展示

为展示不同丢包率对传输的影响，在传输过程中将对传输文件的大小、传输时间以及吞吐率进行计算并展示。通过调节路由器的丢包率展示网络中丢包对数据传输的性能影响

当网络中不发生丢包时，传输一个 180 万字节的数据需要大概 18 秒

图 4.6: 丢包率为 0

当丢包率为 20% 时，虽然丢包率只增加了 20%，但是传输速率却低了将近 30 倍，这是因为在停等机制中只有超时重传，所以一切错误的处理都必须等待固定的秒数（在本实验中我设置的是 200ms）

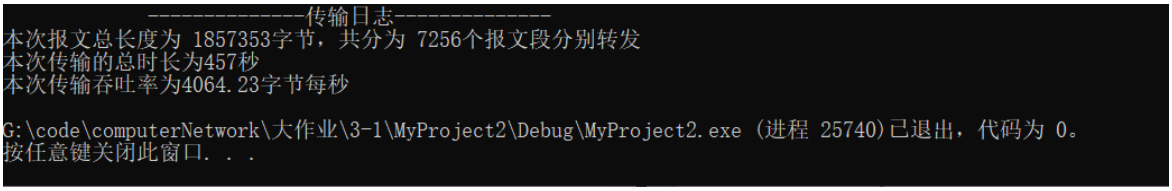

图 4.7: 丢包率为 20%

当丢包率为 50 时，相较于 20 的情况，传输时间稳步增长主要是因为其开销主要花在了等待时间上，传输时间相较于等待时间已经很小了，所以等待时间的翻倍最终也使传输时间翻倍。

图 4.8: 丢包率为 50%

5 辅助函数

除了上面的具体流程外，完成本程序还需要实现很多辅助功能，在本节中将对这些功能进行介绍。

5.1 校验和

首先最终要的就是校验和的计算，它是差错检验的核心途径，具体的校验和计算方法就是传入一个 char* 指针指向需要计算校验和的起始地址以及他的字节长度，根据字节长度计算需要计算多少次校验和。

校验和的具体计算流程为对每一个 16 位数进行二进制相加，如果产生进位就把最高位加到最后一位，等到所有的计算结束后再进行取反。

```
1
2  u_short vericksum(u_short* mes, int size) {
3      int count = (size + 1) / 2;
4      u_short* buf = (u_short*)malloc(size + 1);
5      memset(buf, 0, size + 1);
6      memcpy(buf, mes, size);
7      u_long sum = 0;
8      //buf += 0;
9      //count -= 0;
10     while (count--) {
11         sum += *buf++;
12         if (sum & 0xffff0000) {
13             sum &= 0xffff;
14             sum++;
15         }
16     }
17     return ~(sum & 0xffff);
18 }
19
```

5.2 传输结束通知

当发送端发送完所有的数据报之后，如何通知接收端停止接受呢？在本次实验中，我对标志位设置了一个 OVER 位来通知接收端停止接受消息。当然也要对停止接受的包进行差错检错以及超时重传，具体的部分实现如下：

```
1
2      int endsend() {
3      ALLEND = clock();
4      Header header;
5      char* sendbuffer = new char[sizeof(header)];
6      char* recvbuffer = new char[sizeof(header)];
7
8      header.flag = OVER;
```

```
9       header.checksum = calcksum((u_short*)&header, sizeof(header));
10      memcpy(sendbuffer, &header, sizeof(header));
11  SEND:
12      if (sendto(client, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen) == -1) {
13          cout << "[FAILED] 传输结束信号发送失败...." << endl;
14          return -1;
15      }
16      cout << "[SENDOK] 传输结束信号发送成功...." << endl;
17      clock_t start = clock();
18  RECV:
19      while (recvfrom(client, recvbuffer, sizeof(header), 0, (sockaddr*)&router_addr, &rlen) <=
20          if (clock() - start > MAX_TIME) {
21              if (sendto(client, sendbuffer, sizeof(header), 0, (sockaddr*)&router_addr, rlen) ==
22                  cout << "[FAILED] 传输结束信号发送失败.... 请检查原因" << endl;
23                  return -1;
24              }
25              start = clock();
26              cout << "[ERROR] 传输结束信号反馈超时.... 正在重传" << endl;
27              //goto SEND;
28          }
29      }
30      memcpy(&header, recvbuffer, sizeof(header));
31      if (header.flag == OVER_ACK && vericksum((u_short*)&header, sizeof(header)) == 0) {
32          cout << "[FINFISH] 传输结束消息发送成功.... 感谢使用" << endl;
33          return 1;
34      }
35      else {
36          cout << "[ERROR] 数据包错误.... 正在等待重传" << endl;
37          goto RECV;
38      }
39  }
40
```

6 总结

通过本次实验，我通过对 UDP 的编程方式熟悉了可靠传输的相关知识，实现了基本停等机制的超时重传以及差错检测，完成了 rdt3.0 的基础上还对三次握手。四次挥手进行了改进，为后续流量累计确认以及拥塞控制的编程打下了编程以及知识基础。