上机作业 3 ——
使用 UDP 实现可靠的文件传输

1711430 江玥

2019 12 12

摘 要

基于 **udp** 协议，实现了类 **ftp** 的功能，支持多用户从服务器下载文件，利用滑动窗口协议的方法发送文件内容以及接受文件内容，利用超时重传、**ack** 确认等机制保证了下载文件的可靠性。

目录

1 作业要求及实现

1.1 基本要求

此次作业的的基本要求：

1. 下层使用 **UDP** 协议，即用数据包套接字完成本次作业
2. 分别完成客户端和服务器端的程序
3. 能够实现可靠的文件传输系统，在 **UDP** 协议上实现 **FTP** 协议的可靠性，保证传输的文件是完整且正确的
4. 实现多客户同时向服务器端发出请求，能够同时下载文件。

1.2 具体实现

在本次作业中，我具体实现了如下功能：

1. 此次 **socket** 编程工作是基于 **UDP** 协议实现的
2. 采用多进程的方式实现多客户端同时请求
3. 采用三次握手的方式来建立客户端与服务器的连接
4. 利用结构体对数据段进行封装
5. 利用活动窗口协议、选择性确认的方式实现文件传输的完整性和顺序接收
6. 开启定时器，若在一定时间内未收到 **ack**，则进行重传

2 项目结构简介

2.1 自定义结构体

2.1.1 connection_record

服务器端维护一个用户列表（一个存放 **connection_record** 的 **vector** 容器），来存放进行连接的用户地址等信息：

```
std::vector<connection_record> records;
```

单个连接记录的结构：

```
struct connection_record
{
  struct sockaddr_in client_addr;
  int port;
```

```
  pid_t pid;
};
```

每一个 **connection_record** 记录了一个与服务器进行了连接的客户端的连接情况。**connetion_record** 的结构体成员：

- **client_addr**：对服务器请求连接的客户端的网络地址
- **port**：服务器端分配的与该客户端进行通讯的固定端口号
- **pid**：该通讯进程的进程号

在整个程序中，父进程维护用户列表，子进程来实现对具体用户的文件传输。在进行完文件传输后，父进程对子进程的资源进行回收。

2.1.2 msgSeg

数据被封装为数据段的格式，再进行传输。
具体结构：

```
#define BUFFER_SIZE 1024
struct msgSeg
{
  int seq; // 序列号
  int ack; // ack 的序列号
  int dataSize; // buffer 里存的字节数
  char sign; // 标志位
  char buffer[BUFFER_SIZE]; // 消息存放缓冲区

  msgSeg() {
    seq = 0;
    ack = 0;
    dataSize = 0;
    sign = '';
  }
};
```

BUFFER_SIZE 的大小可进行修改，但需要是 **2** 的次方的数。在 **udp** 协议的传输过程中，需要用 **sendto** 和 **recvfrom** 进行消息的传递，在每次消息交互时，我们先把数据封装为 **msgSeg**，另一端接收到 **msgSeg**，也先对其进行解析，在将数据写入文件中，实现文件的传输。

msgSeg 的结构体成员：

- **seq**：序列号，是保证数据段顺序接收、不缺失的判断依据。

- **ack**：在传输文件时，由客户端发出。若 **ack=n**，则是客户端告诉服务器已经完成前 **n-1** 个数据段的顺序接收了，下一个想要收到的数据段的序列号为**n**。
- **dataSize**：字节数，在传输文件的过程中，若不发生数据段中数据的缺失，则除了最后一个数据段以外，其余的数据段的 **dataSize** 都等于 **BUFFER_SIZE**。
- **sign**：标志位。**'H'**：表示打招呼阶段；**'S'**：表示握手阶段；**'G'**：表示开始进行文件传输的阶段；**'O'**：表示正在进行文件传输的阶段；**'F'**：表示文件传输结束的阶段。
- **buffer**：消息存放缓冲区，要传输的文件内容需要存入 **buffer** 中。

2.1.3 sendingWindow

消息发送窗口：

```
#define SENDINGWINDOW_SIZE 6
struct sendingWindow
{
  int head; // 消息队列的头部下标
  int tail; // 消息队列的尾部下标 +1
  int sendBase; // 消息队列头部的数据段的序列号
  int nextseqnum; // 下一个要进入消息队列的数据段的序列号
  struct msgSeg window[SENDINGWINDOW_SIZE]; // 消息段发送窗口
  sendingWindow()
  {
    head = 0;
    tail = 0;
    sendBase = 0;
    nextseqnum = 0;
  }
};
```

服务器端要向客户端发送文件，因此要有消息发送窗口对已发送但未 **ack** 的数据段进行暂存，比传输单个数据段，再阻塞等待 **ack** 确认后传输下一个数据段的效率要高。在这里设置的发送窗口大小为 **6**，后续可通过实验得到最合适的窗口大小。

sendingWindow 的结构体成员：

- **head**：是消息队列头部的下标，指向发送窗口最先发送却未被 **ack** 的数据段。
- **tail**：是消息队列尾部的下标 **+1**，指向下一个该进行封装的无效数据段。

- **sendBase**：消息队列头部数据段的序列号，是最先发送但未被 **ack** 的序列号（从客户端接收到的最大 **ack** 数）
- **nextseqnum**：下一个要进入消息队列的数据段的序列号
- **window**：**SENDINGWINDOW_SIZE** 个 **msgSeg** 构成的数组

2.1.4 recvingWindow

```
#define RECEIVINGWINDOW_SIZE 6
struct recvingWindow
{
  int recvBase;
  int emptyPos; //未被使用的区域
  struct msgSeg window[RECEIVINGWINDOW_SIZE];
  bool isRecv[RECEIVINGWINDOW_SIZE];
  recvingWindow() {
    recvBase = 0;
    emptyPos = 0;
    for (int i = 0; i < RECEIVINGWINDOW_SIZE; i++)
    {
      isRecv[i] = false;
    }
  }
};
```

客户端要接收从服务器发来的文件数据段，并写入相应文件（完成下载），为了支持选择 **ack** 机制，需要有消息接收窗口。当接收到数据段时，先将其存入接收窗口队列中，等待接收满整个接收队列时，再一起写入文件。这样可以节省写文件的 **io** 时间。在接收到乱序的数据段时，也先存入接收窗口，等待接收完之前所有的数据段再一起写回。

若顺序接收到了前 **n** 个数据段，则发送**ack=n+1**，表示下一次需要接受到的数据段的序列号为 **n+1**。

- **recvBase**：接收队列有效的数据段中最小的序列号
- **emptyPos**：指向下一个未存入有效数据段的空位置
- **window**：接收窗口，**RECEIVINGWINDOW_SIZE** 个 **msgSeg** 构成的数组。
- **isRecv**：接收到有效数据段，在 **isRecv** 的对应下标置为 **true**

2.2 项目组成

项目组成：

```
udp-ftp/
├── send_files
├── recv_files
├── makefile
├── tools.h
├── multiServer.cpp
├── server.cpp
├── client.cpp
├── ftpServer.h
├── ftpServer.cpp
├── ftpClient.h
└── ftpClient.cpp
```

- **send_files**：服务器端用来发送的文件存储位置
- **recv_files**：客户端用来接收的文件存储位置
- **makefile**：项目编译文件
- **tools.h**：引入项目所需头文件，定义项目所需的全局常量和结构体。
- **multiServer.cpp**：提供多客户端支持功能的 **cpp** 文件，维护用户列表的父进程程序。
- **server.cpp**：由父进程 **multiServer** 派生出的子进程程序，在父进程接收到客户端的连接请求后，有子进程服务器进行后续的信息交互过程。
- **ftpServer.h, ftpServer.cpp**：提供类 **FTP** 服务器在文件传输过程中进行的进行操作，分别用函数进行封装。
- **client.cpp**：请求下载文件的客户端程序，对服务器发出请求，接收服务器发来的数据段，并写入相应文件中。
- **ftpClient.h, ftpClient.cpp**：提供类 **FTP** 客户端在文件传输过程中进行的具体操作，分别用函数进行封装。

3 主要代码和功能实现

3.1 用多进程实现多客户端连接

3.1.1 fork 进程相关知识了解

父进程使用**fork** 函数来创建子进程：

1. 在父进程中，**fork** 返回新创建子进程的进程 **ID**
2. 在子进程中，**fork** 返回 **0**
3. 如果出现错误，**fork** 返回一个负值

因此可以看出，通过 **fork** 返回值的不同我们就能区分开父进程和子进程，使父进程和子进程后续分别进行不同的操作。

fork 常用的场景：

1. 一个父进程希望复制自己，使父、子进程同时执行不同的代码段。这在网络服务中是常见的。父进程等待客户端的服务请求，当这种请求到达时，父进程调用 **fork**，使子进程处理此请求。父进程则继续等待下一个服务请求的到达。
2. 一个进程要执行一个不同的程序。这是 **shell** 中常见的情况，子进程从 **fork** 返回后立即调用 **exec**

3.1.2 具体实现逻辑和代码

实现的整体逻辑如下：

1. 在父进程，创建一个数据报套接字 **(udp)**，和本地 **ip** 和 **8888** 端口进行绑定 **(bind)**，用来监听客户端发来的连接请求。

```
    struct sockaddr_in server_addr, client_addr;
    socklen_t len = sizeof(client_addr);
    msgSeg recvSeg, sendSeg;
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    // 用于连接的套接字
    if (fd < 0)
    {
        cout << "create server socket failed!" << endl;
        exit(-1);
    }
    memset(&server_addr, 0, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    server_addr.sin_port = htons(8888); // 用来监听的端口

    if (bind(fd, (struct sockaddr *)&server_addr,
    ↪  sizeof(server_addr)) < 0)
    {
        cout << "server bind failed!" << endl;
```

```
        close(fd);
        exit(-1);
    }
```

2. 一旦接收到来自客户端发来的请求后，判断是否为打招呼请求（数据段的标志位为'H'），如果是的话，进行后续进一步处理。

```
while (true)
    {
        count = recvfrom(fd, &recvSeg, sizeof(recvSeg), 0,
        ↪  (struct sockaddr *)&client_addr, &len);
        // 接收请求
        if (count == -1) // 接收失败
        {
            cout << "ERROR: recv error!\n";
            exit(-1);
        }
        cout << "recv pkg\n";

        if (recvSeg.sign == 'H') // 若是客户端的连接请求（打招
        ↪  呼）
        ......
```

3. 为请求连接的客户端分出一个空闲的服务器端口号，用于子程序中服务器地址的端口号，进行文件传输操作，并且将这个新分配的端口号告诉客户端。

```
if (recvSeg.sign == 'H')
        {
            cout << "recv Hi pkg\n";
            port = getClientPort(client_addr);
            // 看这个客户端是否已经在用户列表中：如果已经与该服
            ↪  务器建立连接，则返回该服务器与其通讯的端口号；否
            ↪  则，返回 0
            // 已经在用户列表中的情况发生在：客户端已经发送 Hi
            ↪  请求，并且服务器也给它分配了通讯的端口号，但它没
            ↪  有收到相应的回复。
            if (port != 0) // 不为 0，说明已经建立连接
            {
                cout << "Is acked client\n";
                sendSeg.seq = port;
                sendSeg.ack = port;
```

```
                sendSeg.dataSize = 0;
                sendSeg.sign = 'H';
                sendto(fd, &sendSeg, sizeof(msgSeg), 0,
                ↪  (sockaddr *)&client_addr, sizeof(*(sockaddr
                ↪  *)&client_addr));
                cout << "resend port: " << port << endl;
            }
            else // 未建立连接，分配新的端口号
            {
                port = getFreePort();
                sendSeg.seq = port;
                sendSeg.ack = port;
                sendSeg.dataSize = 0;
                sendSeg.sign = 'H';
                sendto(fd, &sendSeg, sizeof(msgSeg), 0,
                ↪  (sockaddr *)&client_addr, sizeof(*(sockaddr
                ↪  *)&client_addr));
                // 给客户端回复打招呼信息，并告诉其以后连接用的
                ↪  端口号
```

查找用户连接记录表中是否存在此客户端的函数：

```
int getClientPort(sockaddr_in client_addr) {
    for (size_t i = 0; i < conn_records.size(); i++)
    {
        if (conn_records[i].client_addr.sin_addr.s_addr ==
        ↪  client_addr.sin_addr.s_addr &&
        ↪  conn_records[i].client_addr.sin_port ==
        ↪  client_addr.sin_port &&
        ↪  conn_records[i].client_addr.sin_family ==
        ↪  client_addr.sin_family)
        {
            return conn_records[i].port;
            // 存在此条记录，返回这条记录已经分配好的端口号
        }
    }
    return 0;
    // 不存在则返回 0
}
```

为新连接的客户端分配新端口号的函数：

```cpp
int getFreePort() {
    struct sockaddr_in addr;
    socklen_t len = sizeof(addr);
    int port = 0;
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    // 创建新的套接字，为了返回空闲端口号
    if (fd < 0)
    {
        cout << "create new server socket failed!" << endl;
        exit(-1);
    }
    ... // addr 赋值操作
    if (bind(fd, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
        cout << "socket bind addr failed!" << endl;
        close(fd);
    }
    if (getsockname(fd, (struct sockaddr *)&addr, &len) == 0)
    {
        // 用于获取一个套接字的名字
        port = ntohs(addr.sin_port);
        cout << "get free server port: " << port << endl;
    } // 作用是获取本地空闲端口
    close(fd);
    return port;
}
```

其中，**getsockname**：返回与某个套接字相关联的本地协议地址。

4. 由 **fork** 函数返回值的不同区分父进程和子进程，子进程执行 **server.cpp** 中的程序，与客户端进行文件传输等操作；父进程继续监听客户端的连接，并且维护一个用户连接记录表 **(connection_record)**，来存放用户的连接情况。

```cpp
        pid_t pid = fork();
        if (pid < 0) // 创建子进程失败
        {
            perror("fork error");
            exit(EXIT_FAILURE);
        }
        else if (pid == 0)
        {   // 子进程执行
```

```
            cout << "create new process, port - " << port <<
            ↪ endl;
            char program[30], new_port[10], client_port[10],
            ↪ client_ip[30];
            ... // 进行参数的赋值
            ... // 向新程序传输有用的参数
            execve("./server", argv, envp);
        }
        else // 父进程执行
        {
            connection_record temp;
            temp.client_addr = client_addr;
            temp.pid = pid;
            temp.port = port;
            conn_records.push_back(temp);
            // 建立客户端连接记录
        }
```

5. 利用信号量，在子进程变为僵尸进程时，在父进程中调用相关函数对僵尸进程的资源进行回收，并在用户连接记录表中删除这一记录。

```
    srand((unsigned)time(NULL));
    signal(SIGCHLD, SIGCHLD_Handle);
    // 处理僵尸进程，捕捉 SIGCHLD 信号，并交由 SIGCHLD_Handle 函
    ↪ 数进行处理
```

SIGCHLD_Handle 函数：

```
void SIGCHLD_Handle(int signo) {
    int status, t;
    while ((t = waitpid(-1, &status, WNOHANG)) > 0)
    {
        // 删去记录表的该条记录
        for (int i = 0; i < conn_records.size(); i++)
        {
            if (t == conn_records[i].pid)
            {
                cout << "exit process pid: " << t << ", client
                ↪ ip: " <<
                ↪ ntohl(conn_records[i].client_addr.sin_addr.s_addr)
                << ", server port: " << conn_records[i].port <<
                ↪ endl;
```

```
                conn_records.erase(conn_records.begin() + i);
                break;
            }
        }
    }
}
```

当子进程结束时，操作系统会产生一个 **SIGCHILD** 信号，产生该信号后，相应的处理函数会捕捉到此信号，并执行相应操作。

产生僵尸进程的原因：**os** 保留结束的子进程的状态，尤其是返回值，以供主进程使用。当子进程结束以后，会将返回值传给 **os**，**os** 将该值保存在内存中，需要通过调用函数来读取那块内存中的值。若要销毁僵尸进程，我们需要在父进程中拿到子进程结束时的返回值，让 **os** 回收该进程（子进程）的资源。

waitpid：通过 **WNOHANG** 参数设置函数为非阻塞状态，在子进程未进行收回时父进程也不会被阻塞，会继续执行。

3.2 客户端与服务器建立连接

3.2.1 调用的函数

服务器端在子进程创建新的 **socket** 用于与客户端的信息交互，并且与客户端完成三次握手：

```
    server.establish_socket_server(atoi(argv[1]));
    // 从 muticlientserver 传过来的端口号 port
    server.accept(filename);
```

客户端在与服务器的父进程进行打招呼操作后，继续进行后续的握手操作：

```
    client.establish_socket_client();
    client.establish_connection(argv[2], argv[3]);
    // argv[2]：服务器 ip （127.0.0.1）
    // argv[3]：要下载的文件名
```

3.2.2 具体实现逻辑和代码

TCP 中三次握手的具体过程：

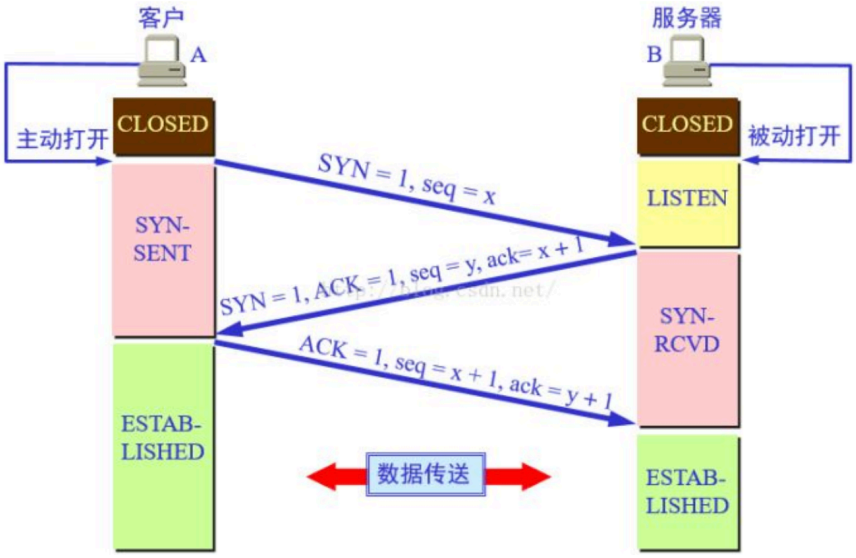

图 1： 三次握手

1. 客户端在发送 H 包之后 (完成打招呼)，也接受到了服务器端发来的 H 包，告诉其以后发送消息使用的新的端口号。

```
if (recvSeg.sign == 'H')
// 收到从服务器响应过来的 H 信息，发来新的端口号，以后使用该端口号进行信息传递
    {
        addr.sin_port = htons(recvSeg.seq);
        // addr => 真实 port  htons
        cout << "recv H new port " << recvSeg.seq << endl;
        break; // if 语句在 while 循环中，此处没有放完整代码
    }
```

之后，客户端进行第一次握手操作，发送标志位为 S 的数据包

```
// 第一个 S 包的发送 与新的端口号建立连接
sendSeg.seq = 0;
sendSeg.sign = 'S';
sendSeg.dataSize = 0;
sendPkg(&sendSeg);
cout << "client send 1th S pkg" << endl;
// 已成功发送第一个 S 包
```

在此阶段，为了防止客户端发送的数据段丢失，导致服务器端无法发送第二个 S 包，在一段时间后若还未收到服务器发来的第二次握手操作，则重发第一个 S 包。

```
while(true) {
        sleep(1);
        sendPkg(&sendSeg);
        ...
    }
```

2. 服务器收到客户端发来的 **S** 包后，进行第二次握手操作，发送第二个 **S** 包

```
            if(recvSeg.sign == 'S') {
                sendSeg.seq = 0;
                sendSeg.ack = recvSeg.seq + 1;
                // 是收到的数据包的下一个 seq
                sendSeg.sign = 'S';
                cout << "server send 2th S pkg\n";
                sendPkg(&sendSeg);
                round += 1; //
             }
```

若服务器发送第二个 **S** 包在发送过程中丢失，此时客户端进行了超时重传第一个 **S** 包，那么服务器会再次发送第二个 **S** 包

```
             else if(recvSeg.sign == 'S') {
                // 向前兼容 应对之前的包丢失
                sendSeg.seq = 0;
                sendSeg.ack = recvSeg.seq + 1;
                sendSeg.sign = 'S';
                cout << "server send 2th S pkg again\n";
                sendPkg(&sendSeg);
             }
```

为了记录是第一次发送第二个 **S** 包还是多次发送，用一个变量 **round** 来记录回合。

服务器发送了第二个 **S** 包，并对客户端发送的第一个 **S** 包进行了 **ack**

3. 客户端接收到了第二个 **S** 包，进行第三次握手操作——发送 **G** 包，表示需要从服务器端下载某个具体的文件，并在发送消息的缓冲区中存储要下载文件的文件名；并对收到的第二个 **S** 包进行 **ack**。

```
    sendSeg.seq = recvSeg.ack;
    sendSeg.ack = recvSeg.seq + 1;
```

```
    sendSeg.sign = 'G'; // get file
    sendSeg.dataSize = strlen(filename);
    strncpy(sendSeg.buffer, filename, strlen(filename));
    sendPkg(&sendSeg);
    cout << "get file " << filename << endl;
```

4. 服务器收到 **G** 包，得知客户端的目的——下载文件，从收到的消息的 **buffer** 中提取出要请求下载文件的文件名，从此，服务器开始使用发送窗口向客户端发送文件内容。

```
            if(recvSeg.sign == 'G') {
                strncpy(filename, recvSeg.buffer,
                ↪ recvSeg.dataSize);
                filename[recvSeg.dataSize] = '\0';
                swindow.sendBase = recvSeg.ack;
                // 客户端下一个想要收到的数据包 seq
                swindow.nextseqnum = recvSeg.ack;
                // 设置发送窗口的 sendBase 和 nextseqnum 的值
                round += 1;
            }
```

3.3 文件传输过程

为了实现传输的可靠性，本次采取了一下方法：

1. 采用滑动窗口协议的方式，在服务器端设置发送窗口，在客户端设置接收窗口。服务器端的发送窗口用来存储已发送但未被 **ack** 的数据段；客户端的接收窗口用来存储已接收但未被写入文件的数据段。

2. 服务器将文件内容封装为一个个数据段，存入发送窗口中，连续发送给客户端，直到发送窗口已满。

   服务器发送完毕后，就等待接收客户端发来的 **ack**，启动超时重传。

3. 客户端接收到数据段后，将其存入接收窗口的相应位置。若遇到乱序的数据包，也都先存入接收窗口中。待接收窗口已满，再一起将服务器发来的文件内容写入相应文件中。

   每接收到一个数据段，客户端就向服务器发送一个 **ack** 消息，**ack** 的数字为客户端想要接收但还未收到的最小序列号的数据段。

4. 乱序处理：客户端将乱序的数据段先存入接收窗口中，但会发送重复 **ack**

5. 数据丢失处理：服务器端若在规定时间内没有收到正确的 **ack**，会认为发生了丢包，进行重传，重传接收窗口中序列号最小的数据包。

3.3.1 服务器端发送文件内容

文件传输的主函数——**fileSendProgram**

```
    sFile = fopen(filename, "rb");
    // 以只读方式打开要传输的文件
    if(sFile == NULL) {
        printf("File open failed\n");
        exit(-1);
    }
    while (sendFinishFlag == false)
    // 传输未完成时，就不断传输文件内容并且接收客户端发来的 ack
    {
        sendFile();
        socketFileAckRecv();
    }
    // 当未接收到所有的 ack 时，继续接收 ack，超时重传
    while (swindow.head != swindow.tail) {
        socketFileAckRecv();
    }
```

具体传输文件的函数——**sendFile** 内调用的 **socketFileSend**（单个数据段的发送）

sendFile:

```
while ((swindow.tail + 1) % SENDINGWINDOW_SIZE != swindow.head &&
 ↪  sendFinishFlag == false) {
        socketFileSend();
    }
    // 发送窗口未满并且文件未发送完毕时，执行 socketFileSend 函数
```

socketFileSend 中对单个数据端进行封装，并调整发送窗口的相关变量

```
    makeNextPkt(&(swindow.window[swindow.tail]));
    // 给 swindow.tail 的数据成员附上值
    if (swindow.tail == swindow.head) // 窗口中只有一个 pkg，且未发送
    {
        swindow.sendBase = swindow.window[swindow.tail].seq;
        // 给 sendBase 赋值，等于发送窗口中头部数据段的序列号
        startTimer(TimeoutInterval);
```

```
        // 启动超时重传机制
    }
    swindow.nextseqnum = swindow.window[swindow.tail].seq + 1;
    printf("send pkg : seq %d dataSize %d sign %c\n",
    ↪ swindow.window[swindow.tail].seq,
    ↪ swindow.window[swindow.tail].dataSize,
    ↪ swindow.window[swindow.tail].sign);
    sendPkg(&(swindow.window[swindow.tail]));
    swindow.tail = (swindow.tail + 1) % SENDINGWINDOW_SIZE;
    // 窗口尾部向后挪一个
```

发送完毕后，根据接收到的 **ack** 消息，改变窗口的相关变量（**head**、**sendBase**），从未判断是否有丢包现象发生

```
   //发送窗口中仍然存在未 ack 的数据包
    while ((swindow.head % SENDINGWINDOW_SIZE) != (swindow.tail %
    ↪ SENDINGWINDOW_SIZE ))
    {
        count = recvfrom(fd, &recvSeg, sizeof(struct msgSeg), 0,
        ↪ (struct sockaddr *)&addr, &len);
        cout << "recv ack: ack " << recvSeg.ack << endl;
        if (recvSeg.ack <= swindow.window[swindow.head].seq)
        // 接收到重复 ack
        {
            cout << "The seqs before seq " << recvSeg.ack << "have
            ↪ been acked" << endl;
        }
        else
        {
            // 对窗口进行“滑动”
            swindow.head += recvSeg.ack - swindow.sendBase;
            swindow.head = swindow.head % SENDINGWINDOW_SIZE;
            swindow.sendBase = recvSeg.ack;
        }
    } // 等待接收完之前的 ack 再继续发
```

若一直未接收到相应的 **ack**，窗口未滑动到正确的位置，则会进行重传——**timeOutHandle**：

```
   if(swindow.head != swindow.tail)
   // 发送窗口中还存在未被 ack 的数据段
   {
```

```
        sendto(fd, &(swindow.window[swindow.head]), sizeof(msgSeg),
        ↪  0, (struct sockaddr *)&addr, sizeof(*(struct sockaddr
        ↪  *)&addr));
        cout << "timeout: sending seq " <<
        ↪  swindow.window[swindow.head].seq << " again\n";
    }
    startTimer(TimeoutInterval);
```

3.3.2 客户端接收文件内容

文件接收的主函数——**fileRecvProgram**

```
    rFile = fopen(str.c_str(), "wb");
    // 以可写入方式要写入的文件
    if(rFile == NULL) {
        cout << "File open failed" << endl;
        exit(-1);
    }
    getFile();
    // 接收文件
```

getFile:

```
    rwindow.emptyPos = 0;
    // 下一个 recvSeg 要存入在接收窗口中的对应下标
    rwindow.recvBase = recvSeg.ack;
    // 接收窗口中最小的未被写入文件的数据段的序列号
    while (recvFinishFlag != true)
    {
        socketFileRecv();
    }
    cout << "file recv complete!\n";
    writeData();
```

初始化接收窗口的相关变量，只要未完成接收，就一直执行 **socketFileRecv** 函数。**socketFileRecv** 中，只要接收到一条消息，就调用 **recvData** 进行处理。

recvData:

```
    if (recvSeg.dataSize != BUFFER_SIZE && recvSeg.sign != 'F')
    // 收到的数据段存在消息缺失
    {
```

```
        cout << "recv incomplete pkg, drop!\n";
        return;
    }
    if (rwindow.recvBase + rwindow.emptyPos == recvSeg.seq)
    // 是最靠前想要收到的数据包（顺序接收）
    {
        strncpy(rwindow.window[rwindow.emptyPos].buffer,
        ↪ recvSeg.buffer, recvSeg.dataSize);
        rwindow.window[rwindow.emptyPos].dataSize =
        ↪ recvSeg.dataSize;
        rwindow.isRecv[rwindow.emptyPos] = true;
        // 表示这个位置的数据段是有效的，已接收到
        while (rwindow.isRecv[rwindow.emptyPos] && rwindow.emptyPos
        ↪ < (RECEIVINGWINDOW_SIZE - 1))
        // 若这个数据段已经在之前乱序接收到
        {
            rwindow.emptyPos++;
        }
        if (rwindow.emptyPos == RECEIVINGWINDOW_SIZE - 1)
        // 接收窗口已满，对接收窗口中的内容整体写入文件中
        {
            cout << "write data" << endl;
            writeData();
            // 写入文件
        }
    }
    else if ((rwindow.recvBase + rwindow.emptyPos < recvSeg.seq) &&
    ↪ (rwindow.recvBase + RECEIVINGWINDOW_SIZE > recvSeg.seq))
    // 乱序收到数据包
    {
        int misorder = recvSeg.seq - rwindow.recvBase;
        strncpy(rwindow.window[misorder].buffer, recvSeg.buffer,
        ↪ recvSeg.dataSize);
        rwindow.window[misorder].dataSize = recvSeg.dataSize;
        rwindow.isRecv[misorder] = true;
        // 表示这个位置的数据段是有效的，已接收到
        // 先存入接收窗口的对应位置中
    }
    else if (rwindow.recvBase + rwindow.emptyPos > recvSeg.seq)
    // 收到已经写入文件的数据包
    {
```

```
40          cout << "recv duplicate pkg!\n";
41      }
42      socketFileAckSend();
43      // 发送 ack
```

需要注意的一点是：在将接收窗口的所有内容写入到文件之后，此时接收窗口中的内容就没有什么意义了，需要将其在 **isRecv** 的对应下标处置为**false**。

发送 **ack——socketFileAckSend**：

```
1       sendSeg.seq = recvSeg.ack;
2       sendSeg.ack = rwindow.recvBase + rwindow.emptyPos;
3       printf("send ack : ack %d\n", sendSeg.ack);
4       sendPkg(&sendSeg);
```

发送接收窗口下一个想要接收到的数据包的序列号。

4 实现效果展示

多客户端：

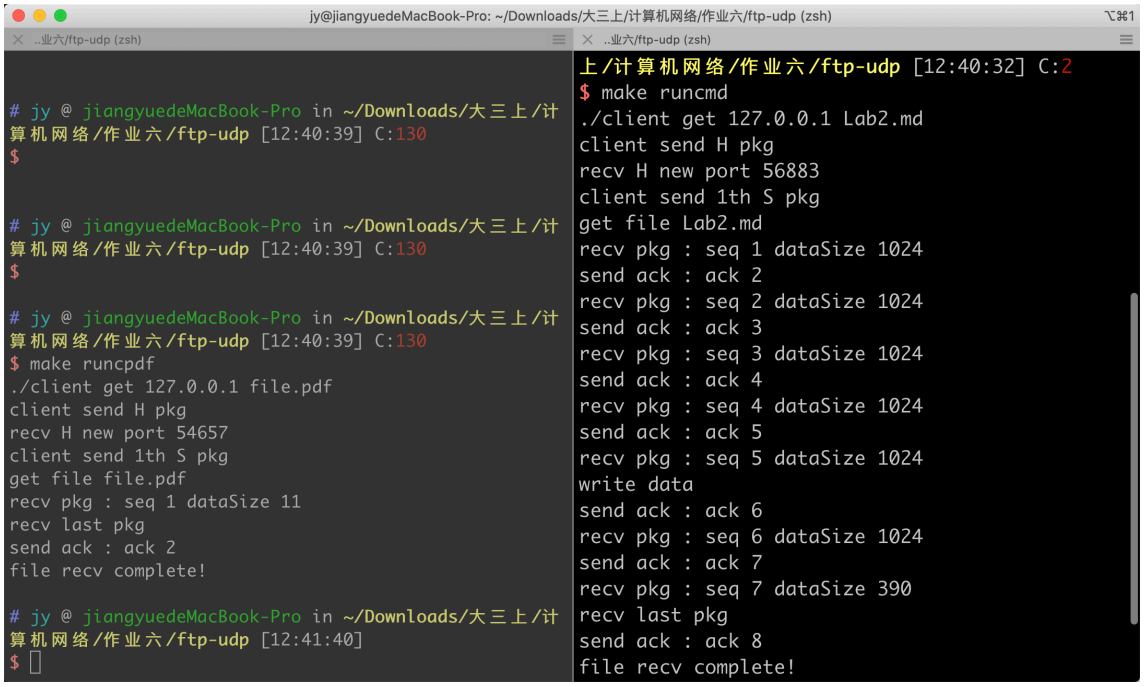

图 2： 多用户下载

make runcpdf：**./client get 127.0.0.1 file.pdf**，表明客户端向服务器请求下载 **file.pdf** 文件

make runcmd：**./client get 127.0.0.1 Lab2.md**，表明客户端向服务器请求下载 **Lab2.md** 文件

服务器端开启多进程处理这两个请求，如下图，可以看出与客户端的信息交互是重叠的，表明同时进行。

```
# jy @ jiangyuedeMacBook-Pro in ~/Downloads/大三上/计算机网络/作业六/ftp-udp [12:41:22]
$ make runs
./multiServer
recv pkg
recv Hi pkg
get free server port: 56883
create new process, port - 56883
recv pkg
recv Hi pkg
get free server port: 54657
create new process, port - 54657
server send 2th S pkg
unacked Pkg : 1
Start timer : 500
send pkg : seq 1 dataSize 1024 sign 0
send pkg : seq 2 dataSize 1024 sign 0
send pkg : seq 3 dataSize 1024 sign 0
send pkg : seq 4 dataSize 1024 sign 0
send pkg : seq 5 dataSize 1024 sign 0
recv ack: ack 2
recv ack: ack 3
recv ack: ack 4
recv ack: ack 5
recv ack: ack 6
unacked Pkg : 6
Start timer : 500
send pkg : seq 6 dataSize 1024 sign 0
```

图 3： 服务器响应信息——1

```
file transport complete !
send pkg : seq 7 dataSize 390 sign F
recv ack: ack 7
recv ack: ack 8
file send complete !
exit process pid: 80201, client ip: 2130706433, server port: 56883
server send 2th S pkg
unacked Pkg : 1
file transport complete !
Start timer : 500
send pkg : seq 1 dataSize 11 sign F
recv ack: ack 2
file send complete !
exit process pid: 80205, client ip: 2130706433, server port: 54657
```

图 4： 服务器响应信息——2

服务器接收到客户端发来的连接请求，给其分配新的端口号，进行完三次握手操作后，进入到文件传输的过程中。

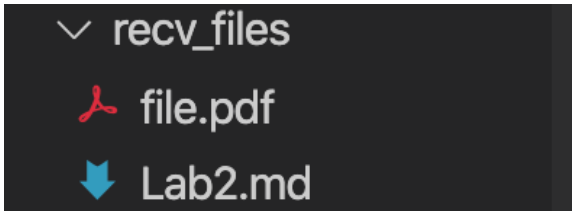

图 5： 下载完成的文件

在命令行中通过 **diff** 命令来判断两个文件的内容是否有出入：

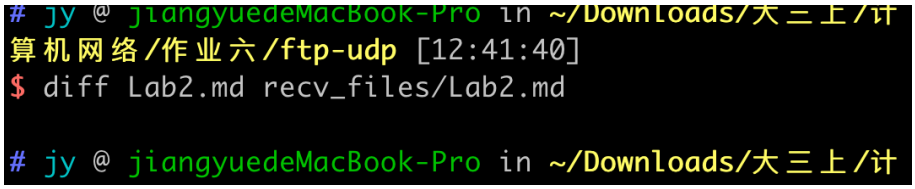

图 6: 比较文件内容异同

没有任何信息输出，表示两文件内容完全相同，证明客户端文件下载成功！