计算机网络传输层报告

马成 20307130112

设计思路

1. 整体协议中客户端和服务端的连接使用的是类似TCP的三次握手的方式，对客户端的虚假请求有一定的防范
2. 可靠传输方面使用的是GBN的方法，在接收方也会有一个缓存区用于应对乱序避免过多的重传
3. 拥塞控制方面使用的是类似TCP Reno的方式，但是我个人对这个方法有一些改动可以在算法细节中详细讨论
4. 断开连接没有做特殊的设计，方法是发现文件传输完成(无论是客户端还是服务端)会发送一个带有结束标志的包，然后关闭连接。另一端接受到这个包之后关闭连接

算法细节

创立连接客户端

1. 大致框架

```
modol='S'
file_name="test2.jpg"
task=FDFTPsocket.Task('client.py')
mytcp=Mytcp((SERVER_IP,CLIENT_PORT),task,file_name,modol)
mytcp.connect()
if modol=='S':
    print(mytcp.getptk())
    mytcp.buf.clear()
    mytcp.stop_timer()
    mytcp.send()
else:
    mytcp.buf.clear()
    mytcp.stop_timer()
    mytcp.recv()
mytcp.task.finish()
os.system("md5sum "+file_name)
```

    1. 规定客户需要的模式和需要的文件建立一个
    2. 清空缓冲区，重置计时器
2. Mytcp.connect()细节

```
def connect(self):
    randseq=random.randint(0,10000)
    ptk=utils.pack('H',randseq,-1,'')
    self.buf.append(ptk)
    self.sendptk(ptk)
    self.start_timer()
    ptk=self.getptk()
    self.buf.clear()
    self.stop_timer()
    self.timeout=3*(time.time()-self.starttime)
```

```
    print(self.timeout)
    self.addr=(self.addr[0],ptk['data'])
    ptk=utils.pack(self.modol,-1,-1,self.file_name)
    self.sendsocket.close()
    self.sendsocket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    self.sendptk(ptk)
    self.start_timer()
    self.buf.append(ptk)
```

1. 想服务器发出一个Hello包尝试建立连接
2. 得到服务器同意分配给这个客户端的端口并重新连接
3. 告诉服务器客户端的具体请求

创立连接服务端

1. 处理客户的第一次握手

```
def testport(PORT):
    presocket=None
    try:
        presocket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        presocket.bind(('172.17.50.166',PORT))
        return presocket
    except:
        if presocket!=None:
            presocket.close()
        return None

def getnewsocket():
    PORT=random.randint(7778,20000)
    presocket=testport(PORT)
    while presocket==None:
        PORT=random.randint(7778,20000)
        presocket=testport(PORT)
    return PORT,presocket

if __name__ == "__main__":
    s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    s.bind((IP, SERVER_PORT))
    while True:
        ptk,addr=s.recvfrom(BUF_SIZE)
        ptk=pickle.loads(ptk)
        if ptk['type']!='H':
            continue
        # 得到一个客户端的请求，选择一个可以使用的端口给这个客户端使用
        PORT,newsocket=getnewsocket()
        thread=MyThread(addr,newsocket)
        thread.start()
        # 将选择的端口告诉请求的客户端
        ptk=utils.pack('B',-1,-1,PORT)
        s.sendto(ptk,addr)
```

1. 通过循环接收客户端的连接请求
2. 通过testport和getnewsocket得到一个可以使用的端口分配给客户端连接

3. 开启一个新的线程并且回传给客户端一个包告诉他分配的端口

2. 专用端口的后续处理

```
def run(self):
    # 接受客户端的第三次握手
    self.mytcp.recvconnect(self.socket)
    print('连接成功')
    if self.mytcp.modol == 'S':
        print('准备接收')
        # 如果是客户端希望传输，这里服务端给客户端一个准备好了的指令
        self.mytcp.sendptk(utils.pack('',-1,-1,'ready'))
        self.mytcp.recv()
        print('已经成功接收')
    else:
        print('准备传输')
        # 如果是客户端希望接收，那么服务端直接开始传输
        self.mytcp.send()
        print('传输完成')
        self.mytcp.timeout=0.2
    self.mytcp.task.finish()
    os.system("md5sum "+self.mytcp.file_name)
```

1. 如果是客户端需要传输，那么先返回给客户一个包表示准备就绪
2. 如果客户希望接收，那么直接开始传输文件

```
def recvconnect(self,socket:socket.socket):
    self.sendsocket=socket
    self.startcon_timer()
    ptk,self.addr=self.sendsocket.recvfrom(self.BUF_SIZE)
    self.timeout=3*(time.time()-self.starttime)
    ptk=utils.unpack(ptk)
    self.stop_timer()
    self.modol=ptk['type']
    self.file_name=ptk['data']
```

3. 注意服务端需要重新获取客户端的地址

接收文件

```
def recv(self):
    self.buf.clear()
    self.buf=[None for i in range(5000)]
    f=open(self.file_name,'wb')
    self.now=0
    finish=False
    while True:
        ptk=self.getptk()
        if ptk['seq']-self.now<5000 and ptk['seq']-self.now>=0:
            self.buf[ptk['seq']-self.now]=ptk
        for i in range(5000):
            if self.buf[i]==None:
                self.buf=self.buf[i:]+[None for j in range(i)]
                break
            self.now+=1
```

```
            ptk=self.buf[i]
            if ptk['type']!='E':
                f.write(ptk['data'])
            else:
                finish=True
            if i==5000-1:
                self.buf=[None for j in range(5000)]
        if self.now%100==0:
            print(self.now)
        ptk=utils.pack('',-1,self.now,'')
        if finish:
            ptk=utils.pack('E',-1,self.now,'')
            self.sendptk(ptk)
            self.sendptk(ptk)
            self.sendptk(ptk)
            break
        self.sendptk(ptk)
    pass
```

```
def startcon_timer(self):
    self.starttime=time.time()
    self.timer=threading.Timer(10,self.closeconn)
    self.timer.setDaemon(True)
    self.timer.start()

def closeconn(self):
    self.sendsocket.close()
    # 一些顾名思义
    print("有坏人客户端欺骗我的感情")
    exit(0)
```

1. 初始化缓冲区，buf是缓冲区，now是当前期待的包的标号
2. 每当收到一个包如果在缓冲区范围内就加入，否则直接丢弃
3. 更新now的值，并且如果找到已经可以写入的包就立刻写入同时就他们移除buf缓冲区
4. 如果收到的是具有结束标记的包，那么结束循环退出
5. 如果超过10s没有连接就直接放弃这个连接
6. 由于关闭连接不可能做到完美，我这里选择了一个比较简单的方法，就是多传这个包很多次，如果非常不幸发送端还是没有收到，那就没办法了

发送文件、可靠传输和拥塞控制

```
def send(self):
    f=open(self.file_name,"rb")
    self.task=FDFTPsocket.Task(self.file_name)
    seq=0
    finish=False
    recvack=recvAck(self)
    recvack.start()
    while True:
        if self.timer==None:
            self.start_timer()
        while seq<self.left+self.Window and finish==False:
            if seq%100==0:
```

```
                print(seq,self.left,self.Window,self.threashold)
            data=f.read(self.MAX_LENGTH)
            if str(data)!="b''":
                ptk=utils.pack('',seq,-1,data)
                self.sendptk(ptk)
                self.buf.append(ptk)
                seq+=1
            else:
                self.sendptk(utils.pack('E',seq,-1,''))
                seq+=1
                finish=True
                break
        if finish:
            break
```

1. 在这个函数其实看上去就是普通的传包，所有的控制全部都在其他的线程中处理。这两个线程一个用于处理超时，一个用于处理从服务端收到的ACK包

```
def start_timer(self):
    if self.timer!=None:
        self.timer.cancel()
    self.starttime=time.time()
    self.timer=threading.Timer(self.timeout,self.timeoutresend)
    self.timer.setDaemon(True)
    self.timer.start()

def stop_timer(self):
    if self.timer!=None:
        self.timer.cancel()
    self.timer=None

def resend(self):
    if self.resending==1:
        return
    # if self.timer==None:
    #     self.start_timer()
    self.resending=1
    # self.stop_timer()
    if len(self.buf):

 print('resend',min(len(self.buf),max(10,self.Window)),utils.unpack(self.buf[0])
['seq'])
    idx=0
    while idx<len(self.buf) and idx<max(10,self.Window):
        self.sendptk(self.buf[idx])
        idx+=1
    self.resending=0
    # self.start_timer()
    pass

def timeoutresend(self):
    print('timeoutresend')
    if self.timer!=None:
        self.stop_timer()
    self.threashold=max(1,int(self.Window/2))
```

```
    self.window=1
    self.resend()
    self.start_timer()
```

2. 这是处理超时的代码细节，timer的设置触发时间就是timeout。
3. 每一次超时将timeout翻一倍，但是处理一下不要让这个timeout过大，尝试过手动设置一些值，但是感觉在复杂的网络下效果一直不好，这个机制就没有使用
4. 拥塞控制上将threashold变为Window的一半，WINDOW变为了1
5. 进行重传，为了不进行过多的重传，一般只允许一个线程进行重传，显然这不是完全的线程安全的，我也并不是要严格的保护，就是一个简单的控制不需要过于严谨
6. 每次我允许至多允许发送WINDOW个，如果buf足够大我希望至少发送10个因为超市一次WINDOW直接变成了1，感觉只传一个有点亏更何况如果重传发生一般就不是只丢一个包可能丢很多的包

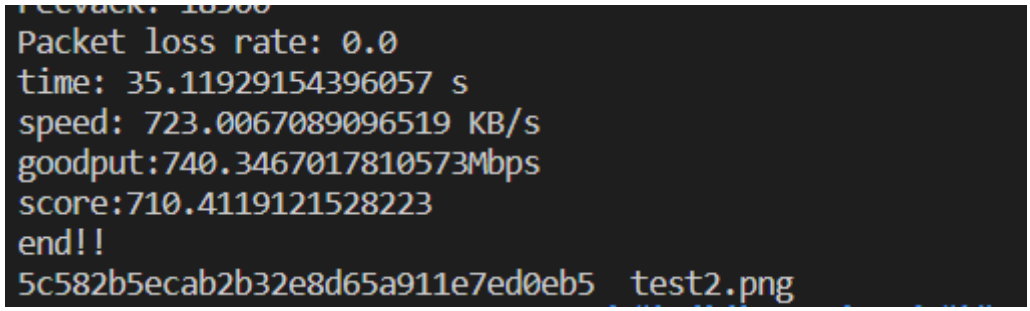
```
class recvAck(threading.Thread):
    def __init__(self,mytcp:Mytcp):
        threading.Thread.__init__(self)
        self.mytcp=mytcp
        self.nowack=-1
        self.numack=0
        self.preack=0
    def run(self):
        while True:
            ptk=utils.unpack(self.mytcp.sendsocket.recv(self.mytcp.BUF_SIZE))
            acknum=ptk['ack']
            if acknum%100==0:
                print('recvack:',acknum)
            left=self.mytcp.left
            if self.mytcp.Window<self.mytcp.threashold:
                self.mytcp.Window+=1
            else:
                self.mytcp.Window+=1/(int(self.mytcp.Window))
            if acknum>left :
                self.mytcp.stop_timer()
                self.numack=0
                self.mytcp.buf=self.mytcp.buf[acknum-left:]
                self.mytcp.left=acknum
                self.mytcp.start_timer()
            else:
                if acknum==left:
                    self.numack+=1
                if self.numack>=3 and acknum!=self.preack:

 self.mytcp.Window=self.mytcp.threashold=max(1,int(self.mytcp.Window/2))
                    self.preack=acknum+11
                    self.numack=0
                    self.mytcp.resend()
                    pass
                if acknum<=self.preack:
                    self.numack=0
            if ptk['type']=='E':
                self.mytcp.stop_timer()
                print('end!!')
                break
```

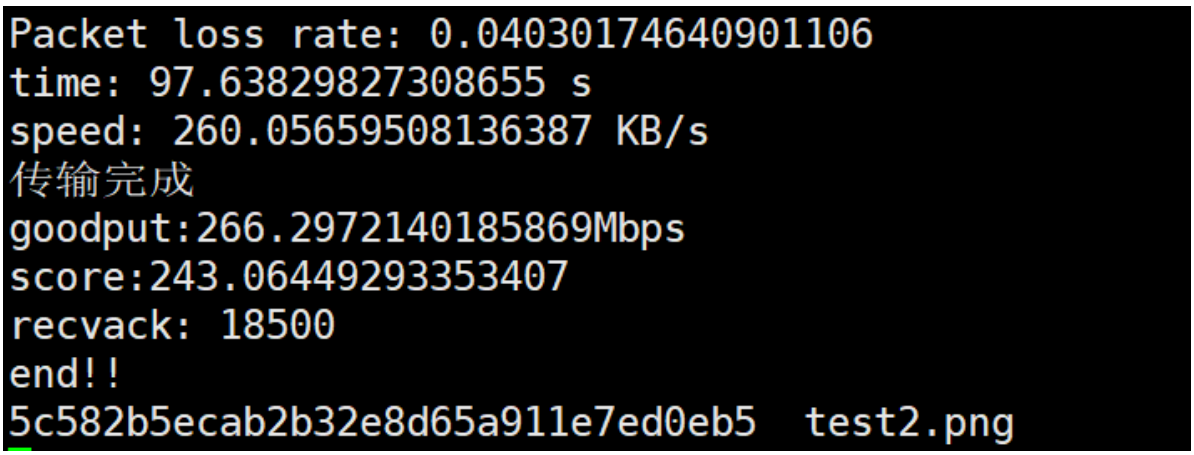
6. 每当接收到一个ACK判断是否是有效的ACK，如果是的话马上更新left的值
7. 如果是重复的ACK进行纪录，达到一定次数之后将WINDOW和threashold都变为window的一半并且重传
8. 书中传统的控制是3次重复ACK就要执行，在校园网上的测试感觉3次就重传特别的浪费，但是一般网络下3次重传感觉效果还行
9. 我大致判断了一下希望一个包不会因为这个机制重传两次（只是粗略的保证，并不严格）

在模拟环境下的性能测试

1. 客户端上传文件
    1. 在比较空闲的校园网环境下ifudanNG.1x速度可以达到1200Kbps，分数大概也在1100~1200分
    2. 在比较忙碌的校园网环境下（使用ifudanNG.1x且对面机房正在上课或使用ifudan.1x）速度可以达到100~400Kbps，分数大概也在100~400分
    3. 3G网络下可能只能达到30kbps左右
2. 客户端下载，感觉速度一般是上传的1/4~1/2左右会明显慢一些



上传



下载

使用手册

1. 将所有的文件拷贝到服务端和客户端，分别运行server.py和client.py
2. 如果server启动的时候发现端口不可用，可以比较简单的让服务端换一个端口，注意要保持两边端口一致
3. 在客户端中选择你的需求，G代表希望获取，S代表希望上传

```
modol='G'
file_name="test2.jpg"
```

4. 没有做特殊的健壮性的处理，请不要上传客户端没有的文件或者下载服务端没有的文件

5. 两端请都在Linux环境下运行
6. 我设置了如果服务器收到第一次握手信号之后10s还没有收到客户端的第二次握手信号就会将socket释放不浪费资源，这是如果客户端再向服务端传请求是无效的。所有如果测试的时候遇到了这个问题，直接关掉客户端再试一次即可，如果网络实在太差可以修改一下这个参数