FTP 协议实验

李雨田
2010012193
计 14

May 23, 2013

Contents

1 实验目的

本实验要求在 Linux 系统上完成一个文件传输协议（FTP）的简单实现。通过本实验，学生不仅可以深入理解 FTP 协议的原理和协议细节，还可以学会利用 Socket 接口设计实现简单应用层协议，掌握 TCP/IP 网络应用程序的基本设计方法和实现技巧。

2 实验内容

2.1 实现方式

本实验分为服务器和客户端两部分，均使用 Python 3 解释器实现。所有操作均按照 RFC959 实现，在测试时先通过与标准 FTP 服务器（pyftpdlib）和客户端（ftplib, The Python Standard Library）测试，保证协议的正确性。本实验中服务器通过多线程方式实现，使得可以同时处理接受多个客户端的请求。由于系统权限问题，端口号并没有严格按照 RFC959 实现。不过这并不影响程序其他部分的运行。

注意到 Python 3 解释器对代码缩进量有要求，不能随便断行。由于页宽有限，以下代码部分断行只是展示需要，实际上必须写成一行。

2.2 服务器实现

服务器需要先监听一个 TCP 端口，对进来的请求分发到不同的线程，实现对不同的客户端分开处理。`FTPServer` 是处理单个客户端请求的类，均在子线程上运行。

```
if __name__ == '__main__':
    listenSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    listenSock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    listenSock.bind((listenAddr, listenPort))
    listenSock.listen(5)
    log('Server started.')
    while True:
        (controlSock, clientAddr) = listenSock.accept()
        FTPServer(controlSock, clientAddr).start()
        log("Connection accepted.", clientAddr)
```

其中`log` 方法实现调试信息的格式化输出。

```
def log(message, clientAddr = None):
    ''' Write log '''
    if clientAddr == None:
        print('\033[92m[%s]\033[0m %s' % (time.strftime(r'%H:%M:%S, %m.%d.%Y
            '), message))
    else:
        print('\033[92m[%s] %s:%d\033[0m %s' % (time.strftime(r'%H:%M:%S, %m
            .%d.%Y'), clientAddr[0], clientAddr[1], message))
```

程序的主框架是一个大的循环，每次根据客户端命令的不同，实现不同的功能。当 Socket 收到的数据是空串，或者命令是QUIT 时，都是客户端断开连接的信号。不过前者是强制断线，后者是按照协议断线。正常断线时应该回复确认短线信息。

```
# ...
if cmd == '': # Connection closed
    self.controlSock.close()
    log('Client disconnected.', self.clientAddr)
    break
log('[' + (self.username if self.authenticated else '') + '] ' + cmd.strip()
    , self.clientAddr)
cmdHead = cmd.split()[0].upper()
if cmdHead == 'QUIT': # QUIT
    self.controlSock.send(b'221 Service closing control connection. Logged
        out if appropriate.\r\b')
    self.controlSock.close()
    log('Client disconnected.', self.clientAddr)
    break
# ...
```

当命令是HELP 时，返回帮助信息，显示服务器支持的命令。

```
# ...
elif cmdHead == 'HELP': # HELP
    self.controlSock.send(b'214 QUIT HELP USER PASS PWD CWD TYPE PASV NLST
        RETR STOR\r\n')
# ...
```

当命令是USER 或者PASS 时，接受客户端传来的用户名和密码。这里并不校验用户名和密码的正确性，设置这个功能有两个目的：第一是方便输出调试信息，在多客户端时清楚看到命令的来源；第二是和一般的 FTP 服务器保持一致，方便检验协议的正确性。如果发现客户端没有登录，将会限制其它功能的使用。

```
# ...
elif cmdHead == 'USER': # USER
    if len(cmd.split()) < 2:
        self.controlSock.send(b'501 Syntax error in parameters or arguments
            .\r\n')
    else:
        self.username = cmd.split()[1]
        self.controlSock.send(b'331 User name okay, need password.\r\n')
        self.authenticated = False
elif cmdHead == 'PASS': # PASS
```

```
    if self.username == '':
        self.controlSock.send(b'503 Bad sequence of commands.\r\n')
    else:
        if len(cmd.split()) < 2:
            self.controlSock.send(b'501 Syntax error in parameters or
                arguments.\r\n')
    else:
        self.controlSock.send(b'230 User logged in, proceed.\r\n')
        self.authenticated = True
# ...
```

当命令是PWD 或者CWD 时，实现与工作目录相关的操作。前者是打印工作目录，后者是更改工作目录。为了实现多客户端分开处理，每个FTPServer 都记录自己的工作目录。更改工作目录的操作通过操作系统接口实现，这样把路径的正确性校验交给操作系统，并且可以根据相应的错误情况输出错误信息。

```
# ...
elif cmdHead == 'PWD': # PWD
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    else:
        self.controlSock.send(('257 "%s" is the current directory.\r\n' %
            self.cwd).encode('ascii'))
elif cmdHead == 'CWD': # CWD
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    elif len(cmd.split()) < 2:
        self.controlSock.send(('250 "%s" is the current directory.\r\n' %
            self.cwd).encode('ascii'))
    else:
        programDir = os.getcwd()
        os.chdir(self.cwd)
        newDir = cmd.split()[1]
        try:
            os.chdir(newDir)
        except (OSError):
            self.controlSock.send(b'550 Requested action not taken. File
                unavailable (e.g., file busy).\r\n')
        else:
            self.cwd = os.getcwd()
            self.controlSock.send(('250 "%s" is the current directory.\r\n'
                % self.cwd).encode('ascii'))
        os.chdir(programDir)
```

当命令是TYPE 时，设置数据传输格式。考虑到绝大多数情况都是使用二进制方式传输，这里只实现了二进制传输格式。使用文本方式传输的话要涉及到某些字符的替换，在实际中并不常用。设置这个功能的目的也在于与一般的 FTP 服务器保持一致，方便检验协议的正确性。

```
# ...
elif cmdHead == 'TYPE': # TYPE, currently only I is supported
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    elif len(cmd.split()) < 2:
        self.controlSock.send(b'501 Syntax error in parameters or arguments
            .\r\n')
    elif cmd.split()[1] == 'I':
        self.typeMode = 'Binary'
        self.controlSock.send(b'200 Type set to: Binary.\r\n')
    else:
        self.controlSock.send(b'504 Command not implemented for that
            parameter.\r\n')
# ...
```

当命令是PASV 时，服务器要进入被动传输模式。服务器监听一个端口，并且把监听的端口返回告诉客户端，并且由客户端来连接。因为监听端口并接受连接是阻塞的，所以在这里将会分发出一个专门的线程进行数据通道的监听与接受。按道理这个命令是在需要数据传输时自动处理的，但是为了清楚地显示出 FTP 协议的流程，这里单独拿出来实现。这里同时也会检查是否已有数据通道建立，或者有正在监听与接受的数据通道，如果有的话会先关闭已有的，保证数据通道正常工作。考虑到当前绝大多数实际应用都是采用被动传输模式，因为防火墙的原因很少使用主动传输模式，所以这里并没有实现PORT。因此在所以数据通道的操作之前，都会检查被动传输模式是否建立，防止程序崩溃。

```
# ...
elif cmdHead == 'PASV': # PASV, currently only support PASV
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    else:
        if self.dataListenSock != None: # Close existing data connection
            listening socket
            self.dataListenSock.close()
        self.dataListenSock = socket.socket(socket.AF_INET, socket.
            SOCK_STREAM, 0)
        self.dataListenSock.bind((self.dataAddr, 0))
        self.dataPort = self.dataListenSock.getsockname()[1]
```

```
        self.dataListenSock.listen(5)
        self.dataMode = 'PASV'
        DataSockListener(self).start()
        time.sleep(0.5) # Wait for connection to set up
        self.controlSock.send(('227 Entering passive mode (%s,%s,%s,%s,%d,%d
            )\r\n' % (self.dataAddr.split('.')[0], self.dataAddr.split('.')
            [1], self.dataAddr.split('.')[2], self.dataAddr.split('.')[3],
            int(self.dataPort / 256), self.dataPort % 256)).encode('ascii')
            )
# ...
```

这就是用来监听数据通道的类。它设置监听 Socket 为超时的，这样它会每隔几秒检测一下监听 Socket 是否关闭。当出现客户端发起被动模式请求，但是没建立连接就断线的情况，服务器就不会一直卡死在监听接受连接的阶段，它会正常跳出。

```
# ...
class DataSockListener(threading.Thread):
    ''' Asynchronously accepts data connections '''
    def __init__(self, server):
        super().__init__()
        self.daemon = True # Daemon
        self.server = server
        self.listenSock = server.dataListenSock
    def run(self):
        self.listenSock.settimeout(1.0) # Check for every 1 second
        while True:
            try:
                (dataSock, clientAddr) = self.listenSock.accept()
            except (socket.timeout):
                pass
            except (socket.error): # Stop when socket closes
                break
            else:
                if self.server.dataSock != None: # Existing data connection
                    not closed, cannot accept
                    dataSock.close()
                    log('Data connection refused from %s:%d.' % (clientAddr
                        [0], clientAddr[1]), self.server.clientAddr)
                else:
                    self.server.dataSock = dataSock
                    log('Data connection accpted from %s:%d.' % (clientAddr
                        [0], clientAddr[1]), self.server.clientAddr)
# ...
```

当命令是NLST 时，返回工作目录的文件列表。这里依然使用了操作系统的接口。注意到 RFC959 要求这个列表通过数据通道传输。

```
# ...
elif cmdHead == 'NLST': # NLST
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    elif self.dataMode == 'PASV' and self.dataSock != None: # Only PASV
        implemented
        self.controlSock.send(b'125 Data connection already open. Transfer
            starting.\r\n')
        directory = '\r\n'.join(os.listdir(self.cwd)) + '\r\n'
        self.dataSock.send(directory.encode('ascii'))
        self.dataSock.close()
        self.dataSock = None
        self.controlSock.send(b'225 Closing data connection. Requested file
            action successful (for example, file transfer or file abort).\r
            \n')
    else:
        self.controlSock.send(b"425 Can't open data connection.\r\n")
# ...
```

当命令是RETR 或STOR 时，分别是从服务器获取一个文件或者存储一个文件到服务器。获取文件的时候使用操作系统的接口，这样对于文件不存在的情况可以很优雅地处理，不会导致程序崩溃。当然对于客户端命令的正确性也有相应的校验，对于数据通道的正确性也有检测，确保不会崩溃。

```
# ...
elif cmdHead == 'RETR':
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    elif len(cmd.split()) < 2:
        self.controlSock.send(b'501 Syntax error in parameters or arguments
            .\r\n')
    elif self.dataMode == 'PASV' and self.dataSock != None: # Only PASV
        implemented
        programDir = os.getcwd()
        os.chdir(self.cwd)
        self.controlSock.send(b'125 Data connection already open; transfer
            starting.\r\n')
        fileName = cmd.split()[1]
        try:
            self.dataSock.send(open(fileName, 'rb').read())
        except (IOError):
```

```
            self.controlSock.send(b'550 Requested action not taken. File
                unavailable (e.g., file busy).\r\n')
        self.dataSock.close()
        self.dataSock = None
        self.controlSock.send(b'225 Closing data connection. Requested file
            action successful (for example, file transfer or file abort).\r
            \n')
        os.chdir(programDir)
    else:
        self.controlSock.send(b"425 Can't open data connection.\r\n")
elif cmdHead == 'STOR':
    if not self.authenticated:
        self.controlSock.send(b'530 Not logged in.\r\n')
    elif len(cmd.split()) < 2:
        self.controlSock.send(b'501 Syntax error in parameters or arguments
            .\r\n')
    elif self.dataMode == 'PASV' and self.dataSock != None: # Only PASV
        implemented
        programDir = os.getcwd()
        os.chdir(self.cwd)
        self.controlSock.send(b'125 Data connection already open; transfer
            starting.\r\n')
        fileOut = open(cmd.split()[1], 'wb')
        time.sleep(0.5) # Wait for connection to set up
        self.dataSock.setblocking(False) # Set to non-blocking to detect
            connection close
        while True:
            try:
                data = self.dataSock.recv(self.bufSize)
                if data == b'': # Connection closed
                    break
                fileOut.write(data)
            except (socket.error): # Connection closed
                break
        fileOut.close()
        self.dataSock.close()
        self.dataSock = None
        self.controlSock.send(b'225 Closing data connection. Requested file
            action successful (for example, file transfer or file abort).\r
            \n')
        os.chdir(programDir)
    else:
        self.controlSock.send(b"425 Can't open data connection.\r\n")
# ...
```

2.3 客户端实现

客户端功能封装在`FTPClient`类里面，通过 Python 3 解释器的交互界面使用，首先需要使用`from client import FTPClient`指令将`FTPClient`类导入。`FTPClient`的整体结构如下所示。

```
class FTPClient():
    def __init__(self):
        # ...
    def parseReply(self):
        # ...
    def connect(self, host, port):
        # ...
    def login(self, user, password):
        # ...
    def quit(self):
        # ...
    def pwd(self):
        # ...
    def cwd(self, path):
        # ...
    def help(self):
        # ...
    def type(self, t):
        # ...
    def pasv(self):
        # ...
    def nlst(self):
        # ...
    def retr(self, filename):
        # ...
    def stor(self, filename):
        # ...
```

`parseReply`方法用来识别服务器的回复。通常情况下是一条命令对应于一条回复。同样这里的 Socket 是会超时的（在`connect`方法里设置），防止服务器意外没有回复导致客户端锁死的情况。这里会自动提取出服务器的回复代码和内容，交回调用者，并且作为调试信息打印出来。

```
# ...
def parseReply(self):
    if self.controlSock == None:
        return
    try:
        reply = self.controlSock.recv(self.bufSize).decode('ascii')
    except (socket.timeout):
```

```
        return
    else:
        if 0 < len(reply):
            print('<< ' + reply.strip().replace('\n', '\n<< '))
            return (int(reply[0]), reply)
        else: # Server disconnected
            self.connected = False
            self.loggedIn = False
            self.controlSock.close()
            self.controlSock = None
# ...
```

`connect` 方法用来连接服务器。按照 RFC959 应该直接连接 21 端口，但是考虑到通用性和系统权限的问题，允许用户指定端口号。连接上服务器之后将 Socket 设置为超时的，防止客户端等待服务器回复进入锁死状态。

```
# ...
def connect(self, host, port):
    if self.controlSock != None: # Close existing socket first
        self.connected = False
        self.loggedIn = False
        self.controlSock.close()
    self.controlSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    self.controlSock.connect((host, port))
    if self.parseReply()[0] <= 3:
        self.connected = True
        self.controlSock.settimeout(1.0) # Timeout 1 second
# ...
```

`login` 方法用来登录服务器。判断用户名正确时再发送密码，完成登录。客户端会记录是否已经登录，如果没有登录，将会限制其他功能的使用。

```
# ...
def login(self, user, password):
    if not self.connected:
        return
    self.loggedIn = False
    self.controlSock.send(('USER %s\r\n' % user).encode('ascii'))
    if self.parseReply()[0] <= 3:
        self.controlSock.send(('PASS %s\r\n' %password).encode('ascii'))
        if self.parseReply()[0] <= 3:
            self.loggedIn = True
# ...
```

quit 方法用来退出服务器，同时将状态初始化，防止退出后再调用其他操作导致错误。

```
# ...
def quit(self):
    if not self.connected:
        return
    self.controlSock.send(b'QUIT\r\n')
    self.parseReply()
    self.connected = False
    self.loggedIn = False
    self.controlSock.close()
    self.controlSock = None
# ...
```

pwd 和cwd 方法用来打印和更改工作目录。这里客户端并不记录工作目录，而是由服务器记录，由用户来判断命令是否成功。因此对于更改工作目录错误等情况能比较简单地处理。

```
# ...
def pwd(self):
    if not self.connected or not self.loggedIn:
        return
    self.controlSock.send(b'PWD\r\n')
    self.parseReply()
def cwd(self, path):
    if not self.connected or not self.loggedIn:
        return
    self.controlSock.send(('CWD %s\r\n' % path).encode('ascii'))
    self.parseReply()
# ...
```

help 方法用来显示帮助信息。

```
# ...
def help(self):
    if not self.connected or not self.loggedIn:
        return
    self.controlSock.send(b'HELP\r\n')
    self.parseReply()
# ...
```

type 和pasv 方法用来更改数据传输格式和模式。设置type 的目的是为了和一般的 FTP 服务器兼容。pasv 方法会拿到服务器提供的地址和端口号，建立数据通道。考虑到当前绝大多数实际应用都是采用被动传输模

式，所以这里只实现了被动传输模式，并且在所有数据通道操作之前都会检测是否已经建立连接，防止出现错误。按道理应该在进行数据传输的时候自动建立数据通道，但是为了展示出 FTP 协议的工作流程，这里单独拿出来实现。

```
# ...
def type(self, t):
    if not self.connected or not self.loggedIn:
        return
    self.controlSock.send(('TYPE %s\r\n' % t).encode('ascii'))
    self.parseReply()
def pasv(self):
    self.controlSock.send(b'PASV\r\n')
    reply = self.parseReply()
    if reply[0] <= 3:
        m = re.search(r'(\d+),(\d+),(\d+),(\d+),(\d+),(\d+)', reply[1])
        self.dataAddr = (m.group(1) + '.' + m.group(2) + '.' + m.group(3) +
            '.' + m.group(4), int(m.group(5)) * 256 + int(m.group(6)))
        self.dataMode = 'PASV'
# ...
```

`nlst` 方法用来打印工作目录的文件列表，按照 RFC959 通过数据通道传输。

```
# ...
def nlst(self):
    if not self.connected or not self.loggedIn:
        return
    if self.dataMode != 'PASV': # Currently only PASV is supported
        return
    dataSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    dataSock.connect(self.dataAddr)
    self.controlSock.send(b'NLST\r\n')
    time.sleep(0.5) # Wait for connection to set up
    dataSock.setblocking(False) # Set to non-blocking to detect connection
        close
    while True:
        try:
            data = dataSock.recv(self.bufSize)
            if len(data) == 0: # Connection close
                break
            print(data.decode('ascii').strip())
        except (socket.error): # Connection closed
            break
    dataSock.close()
    self.parseReply()
```

```
# ...
```

retr 和stor 方法分别用来从服务器获取一个文件或者存储一个文件到服务器。

```
# ...
def retr(self, filename):
    if not self.connected or not self.loggedIn:
        return
    if self.dataMode != 'PASV': # Currently only PASV is supported
        return
    dataSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    dataSock.connect(self.dataAddr)
    self.controlSock.send(('RETR %s\r\n' % filename).encode('ascii'))
    fileOut = open(filename, 'wb')
    time.sleep(0.5) # Wait for connection to set up
    dataSock.setblocking(False) # Set to non-blocking to detect connection
        close
    while True:
        try:
            data = dataSock.recv(self.bufSize)
            if len(data) == 0: # Connection close
                break
            fileOut.write(data)
        except (socket.error): # Connection closed
            break
    fileOut.close()
    dataSock.close()
    self.parseReply()
def stor(self, filename):
    if not self.connected or not self.loggedIn:
        return
    if self.dataMode != 'PASV': # Currently only PASV is supported
        return
    dataSock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    dataSock.connect(self.dataAddr)
    self.controlSock.send(('STOR %s\r\n' % filename).encode('ascii'))
    dataSock.send(open(filename, 'rb').read())
    dataSock.close()
    self.parseReply()
# ...
```

3 思考问题

1. 在 FTP 协议中，为什么要建立两个 TCP 连接来分别传送命令和数据？

   因为命令是有一定的格式的，但是传输的数据可能是任意二进制数据，为了防止把数据误认为是命令，采用两个 TCP 连接来分别传送命令和数据。

2. 主动方式和被动方式的主要区别是什么？为何要设计这两种方式？

   主动方式是服务器主动连接客户端，建立数据通道。被动方式是服务器监听，客户端连接服务器，建立数据通道。区别主要在于发起连接的方向不同。由于有些客户端在防火墙后面，不允许外部主动发起连接，所以设计被动模式。如果开启所有端口，允许外部直接，那样的话对于客户端来说太危险了。当前绝大多是实际应用都是采用被动模式。

3. 当使用 FTP 下载大量小文件的时候，速度会很慢，这是什么缘故？可以怎样改进？

   FTP 使用 TCP 协议传输数据，这样文件之间的界限是无法分辨的，所以每传输一个文件都要建立一条新的连接，传输完成后再关闭连接，表示一个文件的结束。但是 TCP 建立和拆除连接都要三次握手，无疑会增大很多带宽的浪费，导致速度很慢。可以自定一种文件传输格式，标明文件的界限，这样使用一条 TCP 连接也可以传输多个文件；或者使用 UDP 协议传输，但是使用 UDP 协议需要自己进行差错控制，比较麻烦。

4 实验总结

通过这次实验，我理解了 FTP 协议的原理和协议细节，学习了利用 Socket 接口设计实现简单应用层协议，掌握了 TCP/IP 网络应用程序的基本设计方法和实现技巧。