Laboratório 1

CES-35 Redes de Computadores e Internet

15 de Setembro, 2019

Prof. Lourenço A Pereira – ljr@ita.br

Profa. Cecilia de Azevedo Castro Cesar – cecilia@ita.br

Aluno Igor Bragaia, igor.bragaia@gmail.com

Nesse projeto proposto no laboratório 1 da disciplina CES-35 Redes de Computadores e Internet, desenvolveu-se um servidor de transferência de arquivos inspirado no FTP (File Transfer Protocol). A troca de mensagens servidor-cliente e cliente-servidor se dá por meio de bytes utilizando sockets e lógica request-reply e a linguagem escolhida para implementação foi Python3.

A fim de desenvolver uma solução consistente para troca de mensagens, optou-se pela criação de um protocolo simples de comunicação e de uma classe abstrata FTP da qual ambos o cliente FTP e o servidor FTP devem herdar e implementar seus métodos, tais como métodos para conexão, desconexão, e, por fim, um método *run* que torna o socket disponível para enviar e receber mensagens, além de manipular a lógica de envio e recebimento de mensagens, como login e gerenciamento de comandos recebidos.

Classe Message (protocolo de comunicação da aplicação)

As mensagens trocadas entre o servidor e o cliente são instâncias da classe Message, a qual é definida por um método (ls, cd, get ou put, por exemplo) e por um data, que consiste de um dicionário de Python que contém informações complementares sobre o método da requisição.

```
class Message(object):
    def __init__(self, method: str, data: dict):
        self.method = method
        self.data = data
```

Nota-se que a classe Message é muito simples e existe apenas para padronizar a forma como que as mensagens são trocadas entre o servidor e o cliente. O envio de fato de uma Message na forma de bytes pelo socket está definido na classe abstrata FTP.

Classe abstrata FTP

A classe abstrata FTP diz quais métodos o cliente e o servidor, os quais herdam de FTP, devem implementar. Além disso, a classe FTP implementa os métodos send e recv, os quais devem ser usados para enviar e receber mensagens por meio do protocolo desenvolvido para troca de mensagens.

```
from abc import ABC, abstractmethod
import simplejson as json
import socket

BYTES_LEN = 1024*1024  # 1MB

class FTP(ABC):
    def __init__(self):
```

```
        super().__init__()
        self.tcp = None

    @abstractmethod
    def connect(self, address: str):
        pass

    @abstractmethod
    def close(self):
        pass

    @abstractmethod
    def run(self):
        pass

    @staticmethod
    def send(con: socket.socket, message: Message):
        encoded_message = json.dumps(message.__dict__).encode('utf8')
        con.send(encoded_message)

    @staticmethod
    def recv(con: socket.socket) -> Message or None:
        encoded_message = con.recv(BYTES_LEN)
        if not encoded_message:
            return None
        decoded_message = Message(**json.loads(encoded_message.decode('utf8')))
        return decoded_message
```

Nota-se que quando se deseja enviar uma mensagem, executa-se a função **send**, passando como parâmetros uma conexão válida socket e uma instância de Message. Os atributos da instância são então convertidos para uma representação em dicionário, o qual pode finalmente ser serializado como um objeto JSON (JavaScript Object Notation). O resultado dessa serialização é uma representação em formato de string do objeto JSON, o qual é muito semelhante à um dicionário em Python e amplamente utilizado na troca de mensagens na web. Por fim, o JSON serializado é codificado em bytes utilizando UTF-8, o qual pode ser transmitido na conexão socket.

O processo de recebimento de mensagem ocorre de forma semelhante. Inicialmente recebe-se um conjunto de bytes por meio do socket, o qual é decodificado utilizando UTF-8, de forma que é possível desserializar o objeto JSON, onde obtemos um dicionário cujas chaves são os parâmetros a serem fornecidos ao construtor de Message. Por fim, instancia-se uma nova Message, a qual é retornada à aplicação.

Nota-se que a vantagem de se definir a classe Message e os métodos de envios e recebimento de mensagens na classe abstrata FTP é que dessa forma a troca de mensagens é padronizada e deve ser interpretada da mesma forma não importando se a aplicação é um cliente ou servidor.

Classe FTPServer

A classe FTPServer implementa os métodos connect, close e run da classe FTP. Uma vez que o servidor está conectado, a cada conexão socket solicitada por um cliente, uma nova thread é criada para manipular as requisições enviadas por aquele cliente. Dessa forma podemos ter múltiplos clientes acessando o servidor FTP ao mesmo tempo.

Além disso, as mensagens enviadas pelo servidor sempre são instâncias de Message em que o parâmetro data consiste de uma string path (semelhante à um pwd, o

que permitirá o cliente saber em que pasta ele se encontra do cliente; se o cliente não foi autenticado, o path será uma string vazia), uma string text (no caso em que se deseja enviar um texto ao cliente), uma string file (contém o conteúdo do arquivo no caso em que se deseja enviar um arquivo), e uma string filename (contém o nome do arquivo no caso em que se deseja enviar um arquivo). Essas messages, por sua vez, são instanciadas pelo método make_message definido no FTPServer.

```
@staticmethod
def make_message(path: str, base_path: str, auth: str, text='', file='', filename=''):
    return Message('response', {
        'path': '~/{0}'.format(os.path.relpath(path, base_path)) if auth == 'AUTHENTICATED' else '',
        'text': text,
        'file': file,
        'filename': filename
    })
```

Nota-se que quando o servidor recebe uma mensagem de um cliente, o comando já está parseado na forma de uma Message, de forma que um diferente processamento é feito a depender do parâmetro method do objeto Message. Ao fim de cada processamento de requisição do cliente, uma diferente Message de resposta é enviada ao cliente, a depender do comando solicitado.

Destaca-se o fato de o servidor guardar o estado da comunicação com dado cliente. Isso é utilizado, por exemplo, para permitir a execução de diferentes comando caso o usuário esteja devidamente autenticado, por exemplo.

Classe FTPClient

A classe FTPClient implementa os métodos connect, close e run da classe FTP. O cliente FTP também guarda o estado da comunicação com dado servidor. O cliente entende que o usuário ainda não está autenticado caso o path fornecido pelo servidor, que deveria ser uma resposta semelhante à um comando pwd, seja uma string vazia. Se o usuário não esteja autenticado ainda, mensagens de autenticação são trocadas, caso contrário comandos FTP podem ser trocados corretamente.

Além disso, o parsing dos comando executados pelo usuário é realizado pela manipulação da string digitada. Uma vez identificado e comando e sua validade, uma Message de requisição é criada para o comando e então enviada ao servidor. Por fim, a troca de mensagens acontece de forma similar ao FTPServer utilizando os mesmos métodos para envio e recebimento de mensagens presente na classe abstrata FTP.

Leitura e escrita de arquivos

Quando se deseja transmitir um arquivo, o arquivo é lido em formato binário e o conjunto de bytes é adicionado à Message na chave file do dicionário existente no parâmetro data do objeto Message. A fim de ilustar esse procedimento, observa-se o trecho de código em que o servidor executa um comando GET:

```
# get <dirname>
elif request.method == GET:
    filename = os.path.realpath(os.path.join(path, request.data['filename']))
    filename_relpath = os.path.join('files', os.path.relpath(filename, base_path))
    if os.path.isfile(filename):
        with open(filename_relpath, 'rb') as file:
            encoded_file = base64.b64encode(file.read())
            message = self.make_message(path, base_path, status, file=encoded_file,
```

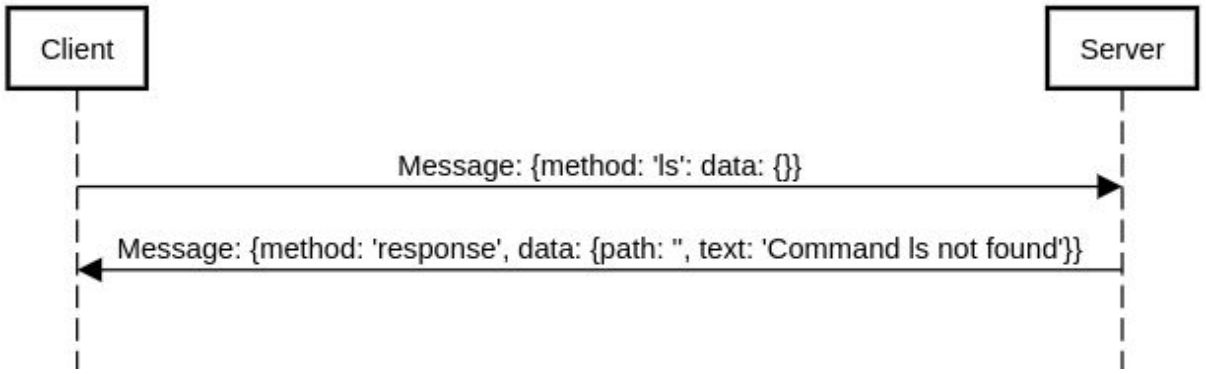
```
                                    filename=request.data['filename'].split('/')[-1])
else:
    message = self.make_message(path, base_path, status, text='No such file or directory')
```

Exemplos de diagrama de troca de mensagens entre o cliente e o servidor

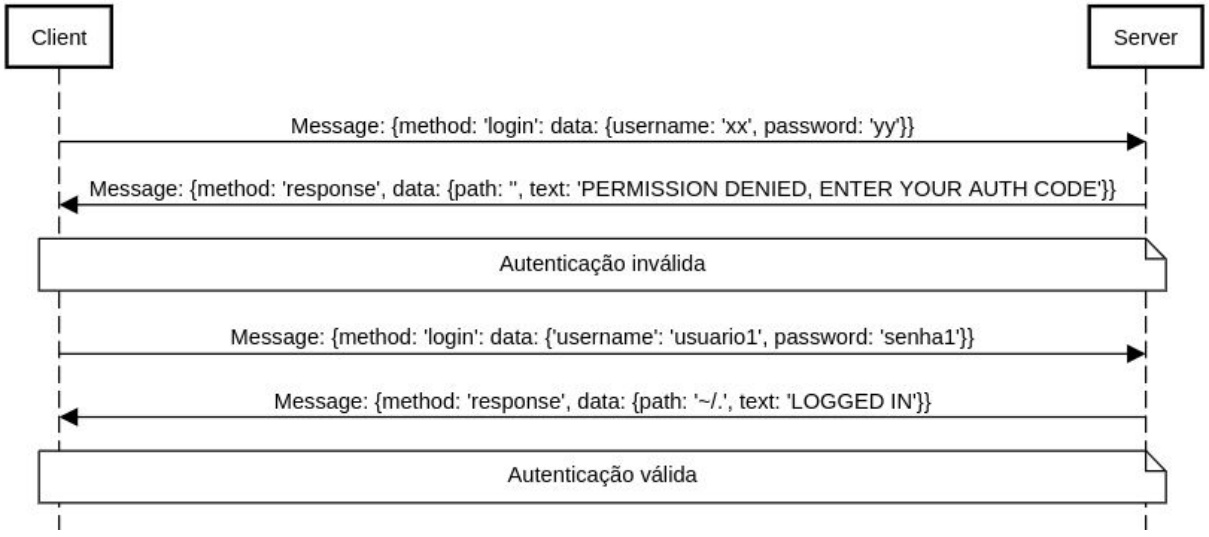
- **Conexão à um servidor**

Quando se deseja conectar à um servidor, o usuário deve digitar “open 127.0.0.1:2121”, por exemplo. Uma vez conectado, inicia-se a etapa de troca de mensagens a seguir.

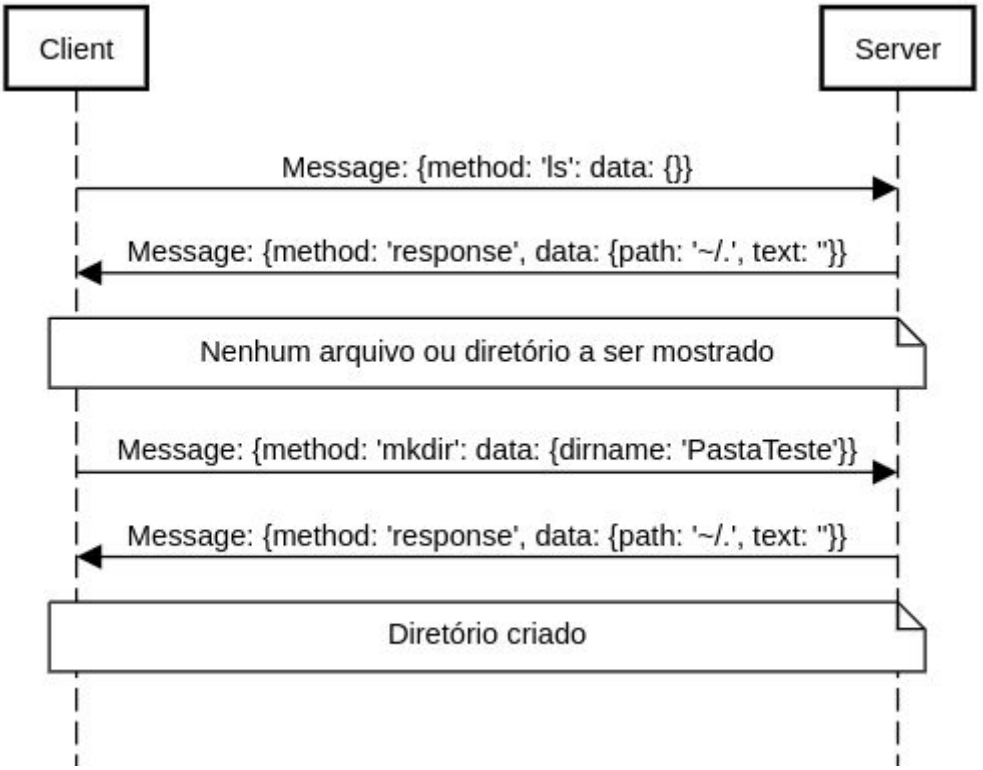
- **Comando inválido para um usuário não autenticado**



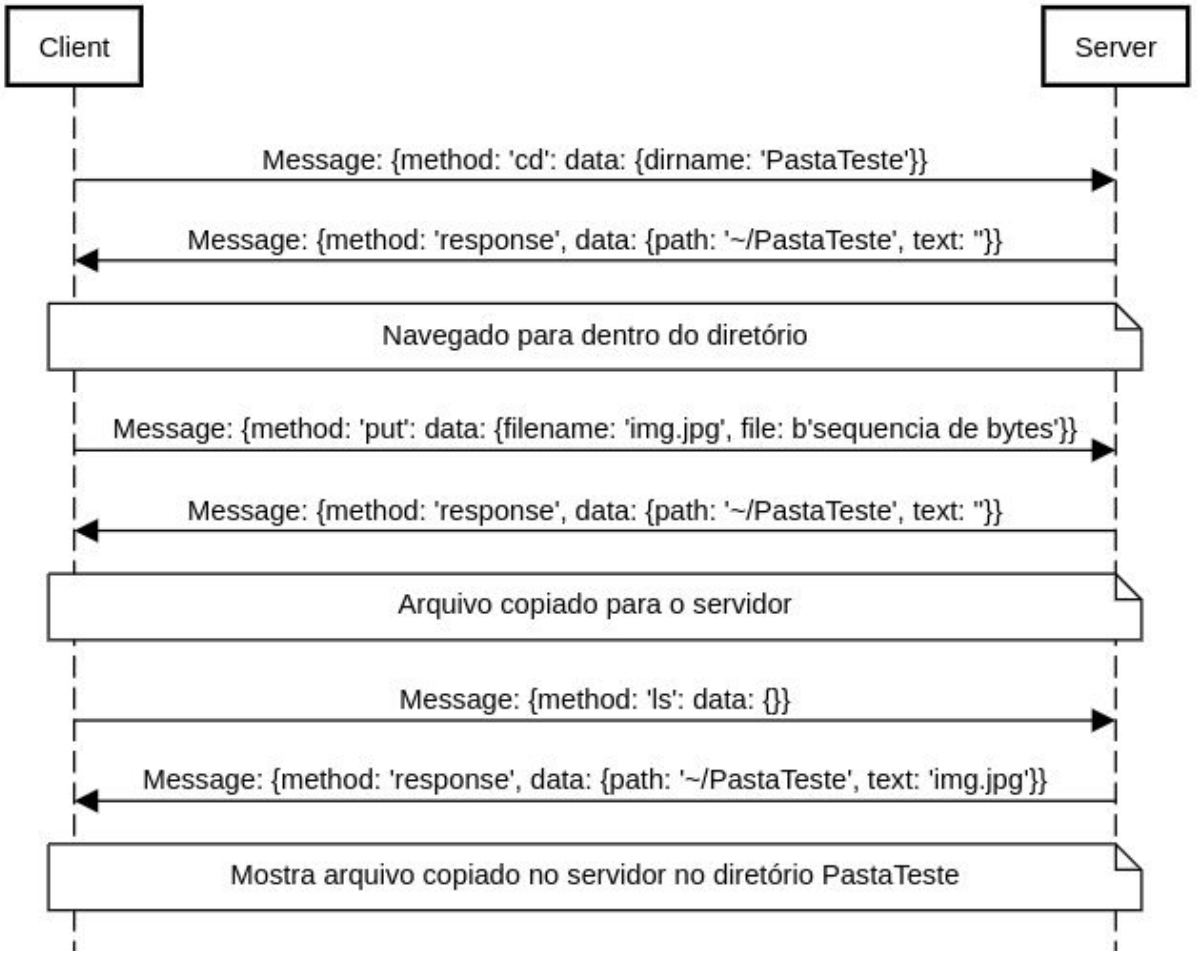
- **Autenticação**



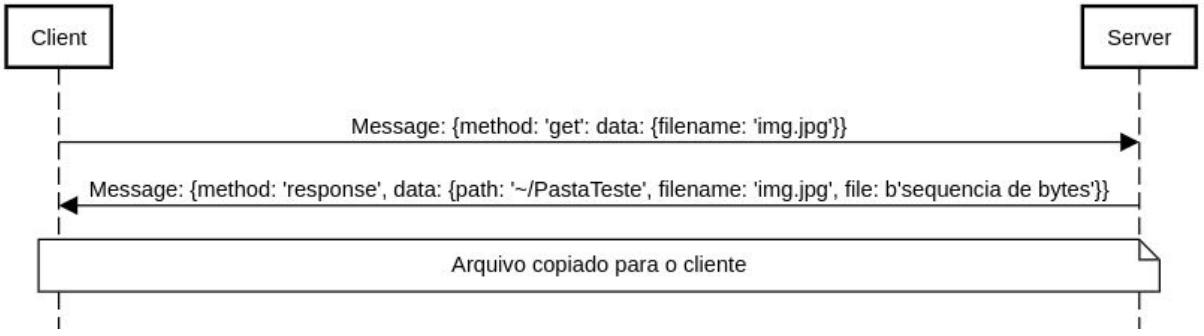
- **Comando ls e comando mkdir para criação de novo diretório**



- **Comando cd, put e ls para navegar para diretório, copiar arquivo e mostrar o arquivo no diretório**



- **Comando get para copiar arquivo do servidor para o cliente**



Com os exemplos acima de diagramas de troca de mensagens entre o servidor e o cliente, pode-se entender o fluxo de troca de dados da aplicação. A fim de ilustrar o fluxo completo de navegação no servidor FTP, seguem screenshots de exemplo.

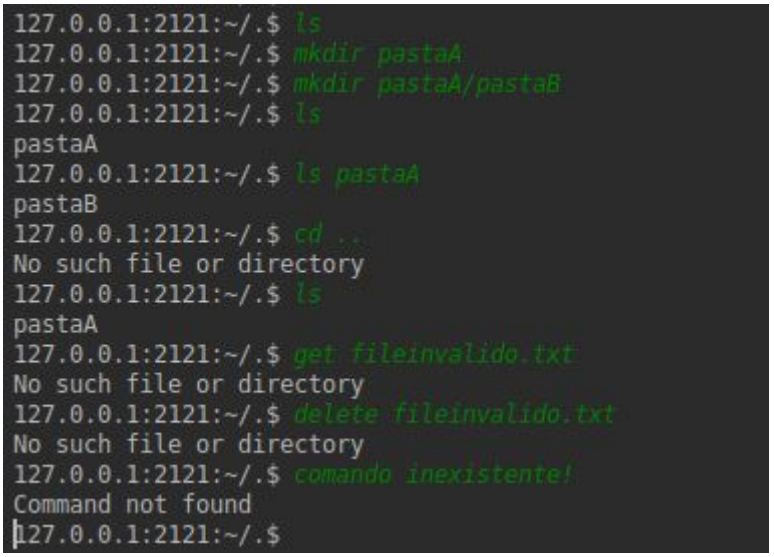
- Conexão ao servidor pelo cliente

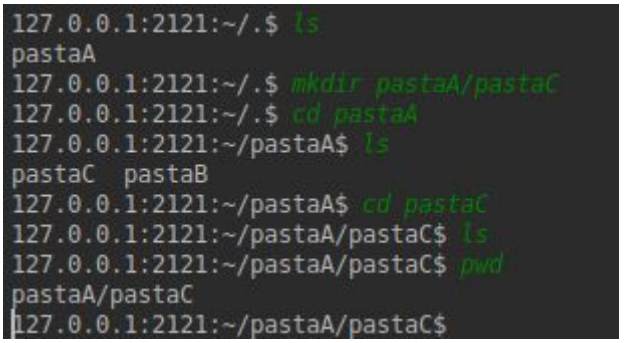
```
        ##############################################################################################
        # FTP CLIENT                                                                                 #
        # Developed by Igor Bragaia (https://igorbragaia.info) using Python3                         #
        # Network classes, ITA 2019.2 - Prof. Lourenço Alves Pereira Jr (https://ljr.github.io)      #
        ##############################################################################################
        # For help, run                                                                              #
        # $ help                                                                                     #
        ##############################################################################################

$ ls
Command ls not found
$ open 127.0.0.1:2122
Connection refused
$ open 127.0.0.1:2121
ENTER YOUR AUTH CODE
username: igor
passowrd: bragaia
PERMISSION DENIED
ENTER YOUR AUTH CODE
username: usuario1
passowrd: senha1
LOGGED IN
127.0.0.1:2121:~/.$
```

- Criação de diretório, navegação em pastas get e put de arquivos

```
127.0.0.1:2121:~/.$ mkdir PastaTeste
127.0.0.1:2121:~/.$ ls
PastaTeste
127.0.0.1:2121:~/.$ cd PastaTeste
127.0.0.1:2121:~/PastaTeste$ put /home/igorbragaia/joao.jpeg
127.0.0.1:2121:~/PastaTeste$ ls
joao.jpeg
127.0.0.1:2121:~/PastaTeste$ get joao.jpeg
127.0.0.1:2121:~/PastaTeste$ cd ..
127.0.0.1:2121:~/.$ ls
PastaTeste
127.0.0.1:2121:~/.$ ls PastaTeste
joao.jpeg
127.0.0.1:2121:~/.$ delete joao.jpeg
No such file or directory
127.0.0.1:2121:~/.$ delete PastaTeste/joao.jpeg
127.0.0.1:2121:~/.$ ls PastaTeste
127.0.0.1:2121:~/.$ ls
PastaTeste
127.0.0.1:2121:~/.$ rmdir PastaTeste
127.0.0.1:2121:~/.$ ls
127.0.0.1:2121:~/.$
```

- Mais alguns comandos para ilustrar o correto funcionamento do projeto de acordo com as solicitações do professor, considerando manipulação correta de erros


127.0.0.1:2121:~/.$ ls
127.0.0.1:2121:~/.$ mkdir pastaA
127.0.0.1:2121:~/.$ mkdir pastaA/pastaB
127.0.0.1:2121:~/.$ ls
pastaA
127.0.0.1:2121:~/.$ ls pastaA
pastaB
127.0.0.1:2121:~/.$ cd ..
No such file or directory
127.0.0.1:2121:~/.$ ls
pastaA
127.0.0.1:2121:~/.$ get fileinvalido.txt
No such file or directory
127.0.0.1:2121:~/.$ delete fileinvalido.txt
No such file or directory
127.0.0.1:2121:~/.$ comando inexistente!
Command not found
127.0.0.1:2121:~/.$


127.0.0.1:2121:~/.$ ls
pastaA
127.0.0.1:2121:~/.$ mkdir pastaA/pastaC
127.0.0.1:2121:~/.$ cd pastaA
127.0.0.1:2121:~/pastaA$ ls
pastaC pastaB
127.0.0.1:2121:~/pastaA$ cd pastaC
127.0.0.1:2121:~/pastaA/pastaC$ ls
127.0.0.1:2121:~/pastaA/pastaC$ pwd
pastaA/pastaC
127.0.0.1:2121:~/pastaA/pastaC$