Laboratório 2 - Implementação de Sistema de Transferência de Arquivos usando Sockets TCP

CES-35 - Redes de Computadores e Internet

Prof. Lourenço Alves Pereira Júnior

Diego T. B. Lima[1]

diegoblima2010@gmail.com

13 de setembro de 2019

[1] Aluno de Graduação em Engenharia de Computação no Instituto Tecnológico de Aeronáutica - ITA. Praça Marechal Eduardo Gomes, nº 50. CEP. 12228-900 - São José dos Campos - SP, Brasil.

1 Introdução

Esse laboratório consiste na implementação de um sistema de transferência de arquivos baseado no protocolo FTP (*File Transfer Protocol*) usando *sockets* TCP para conectar clientes e servidores e transferir *streams* de *bytes*. O objetivo é que clientes possam conectar-se e autenticar-se nos servidores para manipular arquivos remotamente, enviando, recebendo e deletando arquivos. Foi escolhida a linguagem C para implementação dos programas. O sistema foi desenvolvido no Ubuntu 18.04. Os códigos elaborados podem ser visualizados no repositório https://github.com/diegotbl/ftp.

2 Mensagens Trocadas, Diagramas de Sequência e Descrição do Protocolo

2.1 Estabelecimento Inicial da Conexão

O início da conexão dá-se por meio da execução do comando `open <nome_do_servidor>` pelo cliente. Nesse momento o nome do servidor é resolvido através do comando `getaddrinfo` e a conexão é estabelecida com `connect` no *socket*. Em seguida, o usuário insere o nome de usuário que é enviado ao servidor e digita a senha que também é encaminhada ao servidor. Esse processo pode ser visto na Figura 1.

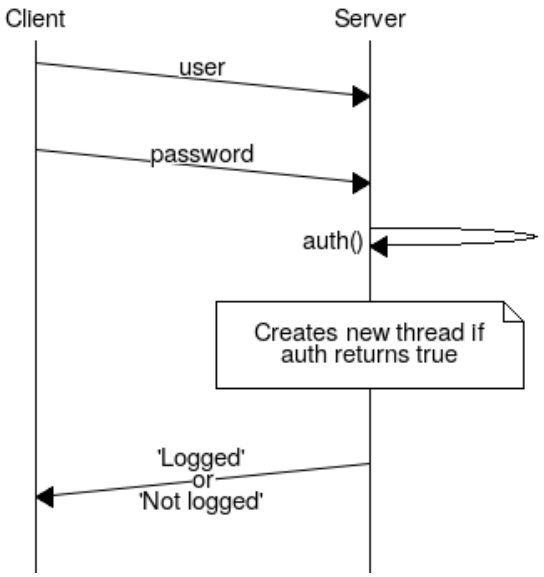


Figura 1: Diagrama de sequência no início da conexão

O servidor realiza o processo de autenticação validando o par usuário/senha, de acordo com as entradas do arquivo *credentials.txt*. Um exemplo de possível arquivo está na Figura 2.

```
1  diego pswd
2  user creativity
3  test huehue
4  another_user another_password
5
```

Figura 2: Exemplo de arquivo *credentials.txt*

Um exemplo de autenticação pode ser visto na Figura 3.

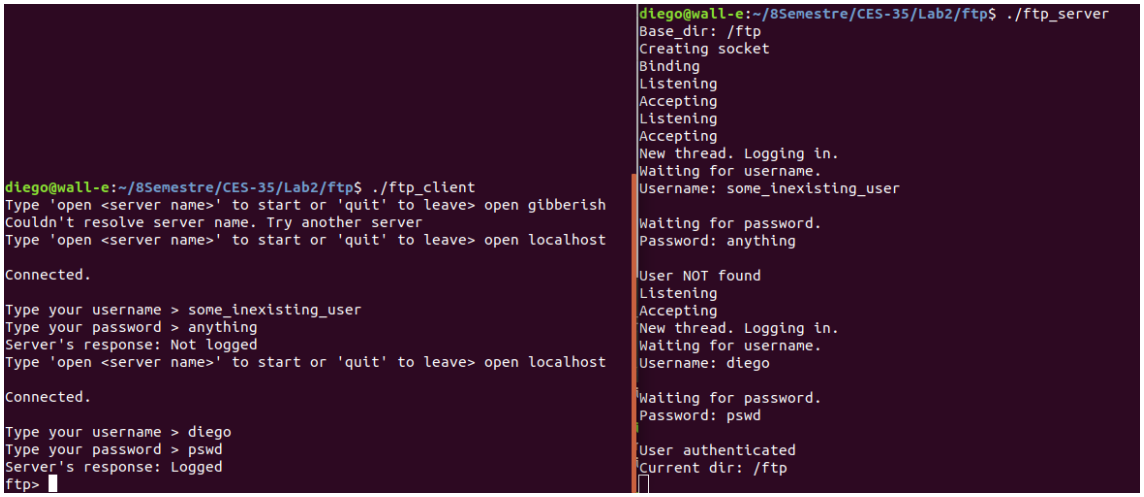
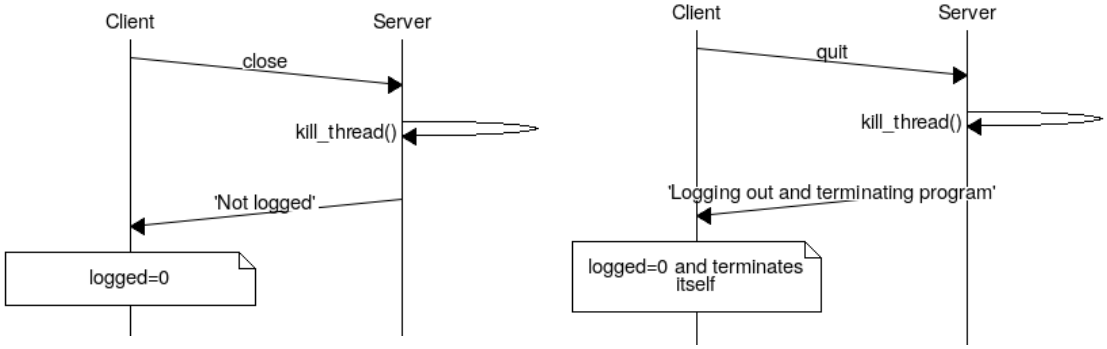


Figura 3: Exemplo de `open`. Cliente a esquerda e servidor a direita.

2.2 Fim de Sessão e Fim de Programa

O comando `close` encerra a sessão atual, pedindo que o usuário execute um novo `open` e o `quit` termina a sessão e encerra do programa. Os diagramas podem ser vistos na Figura 4.



(a) close

(b) quit

Figura 4: `close` e `quit`

2.3 `cd`, `ls` e `pwd`

Os diagramas de sequência com as respectivas mensagens dos comandos `cd`, `ls` e `pwd` estão mostrados nas Figuras 5, 6 e 7.

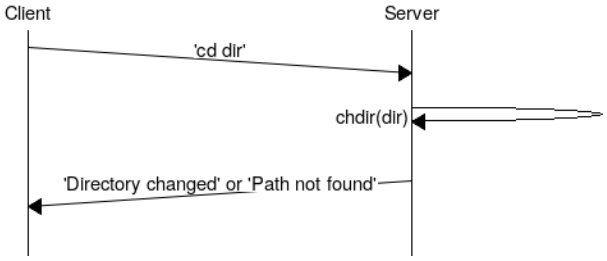


Figura 5: Diagrama de sequência do comando `cd`

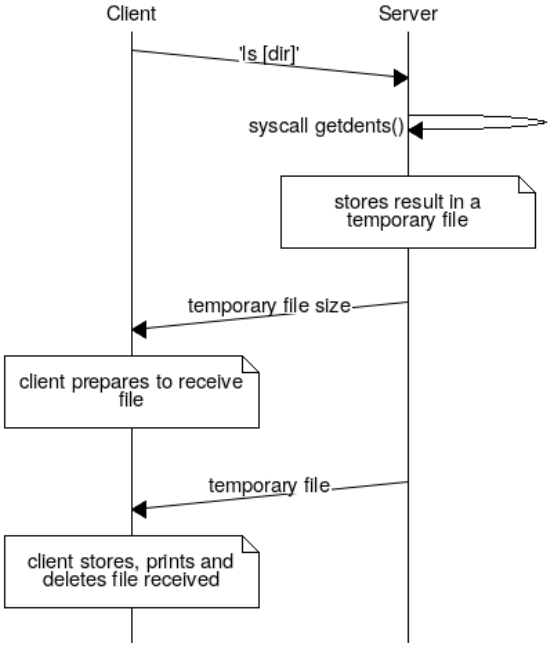


Figura 6: Diagrama de sequência do comando `ls`

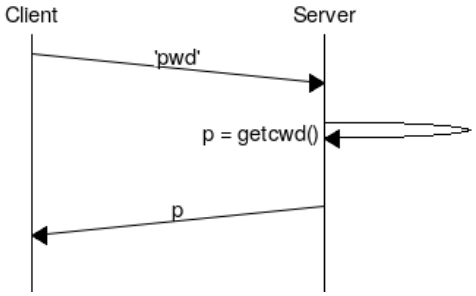


Figura 7: Diagrama de sequência do comando `pwd`

Um exemplo pode ser visto na Figura 8.

```
ftp> ls                                    Command sent from client: ls
file5.txt                                  Command sent from client: ls test
test                                       Command sent from client: cd test
ftp> ls test                               Command sent from client: pwd

ftp> cd test
Server's response: Directory changed
ftp> pwd
Server's response: /ftp/test
ftp>
```

Figura 8: Exemplo dos comandos ls, cd e pwd

2.4 `get`, `put` e `delete`

Os diagramas dos comandos `get`, `put` e `delete` estão nas Figuras 9, 10 e 11.

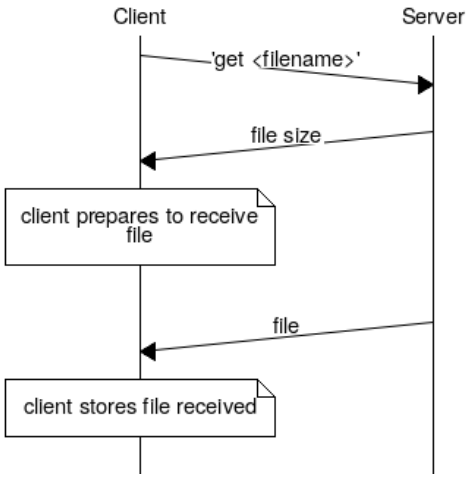


Figura 9: Diagrama de sequência do comando `get`

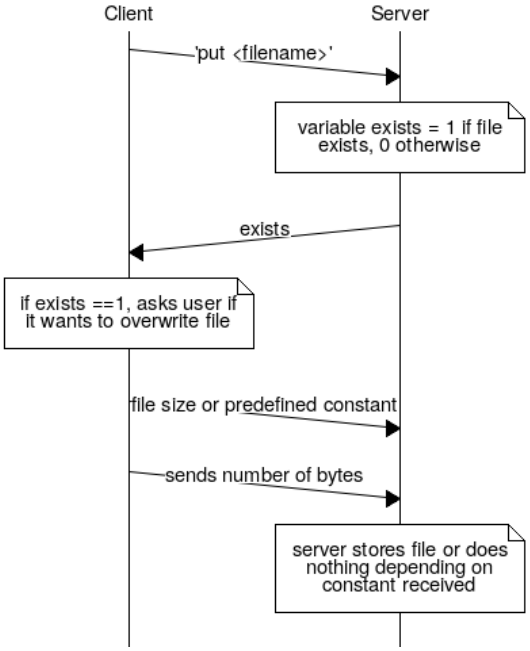


Figura 10: Diagrama de sequência do comando `put`

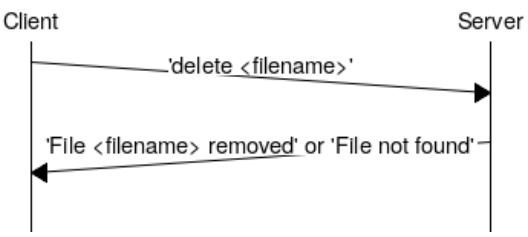


Figura 11: Diagrama de sequência do comando `delete`

Um exemplo pode ser visto na Figura 12.

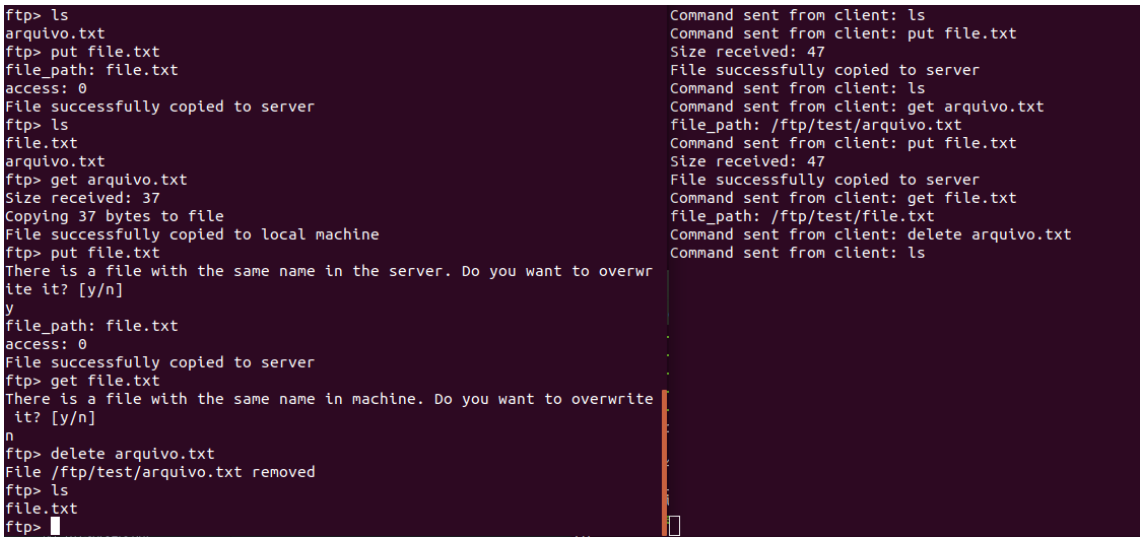


Figura 12: Exemplo dos comandos get, put, delete

2.5 `mkdir` e `remdir`

Os diagramas dos comandos `mkdir`, `rmdir` estão nas Figuras 13 e 14.

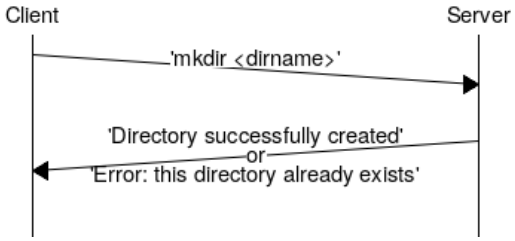


Figura 13: Diagrama de sequência do comando `mkdir`

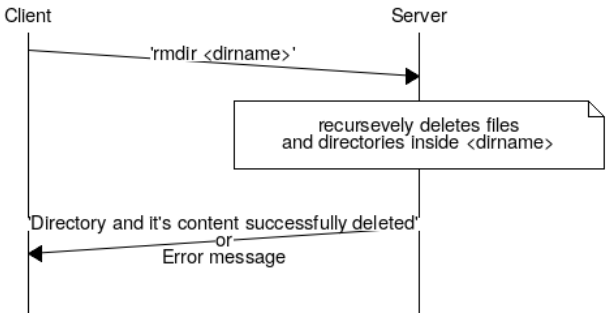


Figura 14: Diagrama de sequência do comando `rmdir`

Um exemplo pode ser visto na Figura 15.

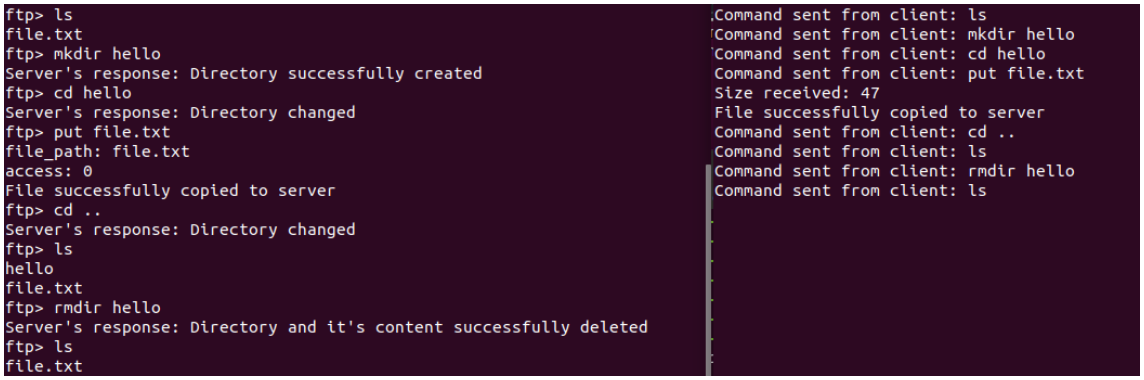


Figura 15: Exemplo dos comandos mkdir e rmdir

3 Gerenciamento de conexões

Para cada usuário que faz *login* no sistema é criada uma nova *thread* que contém variáveis relativas ao contexto dele, tais como o *socket* que será usado para comunicação com ele, o diretório base (expecificado pelo arquivo *base_ dir.txt*) e o diretório atual (que é alterado conforme o usuário vai navegando pelo servidor).

Um problema que surge é que ao alterar o diretório no servidor para um cliente (ao usar *cd* por exemplo), o caminho é alterado para todos. Para contornar isso, fez-se o artifício de realizar uma mudança de diretório a cada comando para o usuário que quiser executar esse comando. Assim sempre o comando será executado no diretório atual do usuário que pediu.

4 Alocação de Memória

Os blocos de bytes que não são muito grandes são recebidos e armazenados em arrays de `char` comuns de tamanho não tão elevado e a informação é manipulada adequadamente.

No caso de *bytes* provenientes de arquivos, primeiro é enviada uma mensagem contendo o tamanho em *bytes* do arquivo a ser transmitido e em seguida aloca-se um `char *` de tamanho adequado para suportar o fluxo de *bytes*, como mostram os códigos 1 e 2.

```
1  void get(char * ptr, int sock_fd, char * my_path){
2      struct stat obj;
3      long size;
4      int filehandle;
5
6      // file_path is assembled with my_path.
7
8      if(access(file_path, F_OK) != -1){  // file exists
9          stat(file_path, &obj);
10         size = obj.st_size;
11         send(sock_fd, &size, sizeof(long), 0);   // send size info to client
12         filehandle = open(file_path, O_RDONLY);
13         sendfile(sock_fd, filehandle, NULL, size);    // send file to client
14     }
15
```

Código 1: Manipulação de memória no comando get no servidor. Versão simplificada do código

```
1  void get_file_response(int sock, char * param){
2      int k, filehandle;
3      char * f;
4      long size;
5
6      recv(sock, &size, sizeof(long), 0); // receive file size
7      printf("Size received: %ld\nCopying %ld bytes to file\n", size, size);
8
9      // file_path is assembled with param.
10
11     filehandle = creat(file_path, O_WRONLY);
12     if(size != 0){
13         // dynamically allocating buffer to receive file
```

```
14              f = (char *)malloc(size*sizeof(char));
15              recv(sock, f, size, 0);
16              // write information read to a new file in the client side
17              k = write(filehandle, f, size);
18              error(k, -1, "Reading failed.\n");
19              close(filehandle);
20              printf("File successfully copied to local machine\n");
21          }
22      }
23
```

Código 2: Manipulação de memória no comando get no cliente. Versão simplificada do código

5 Parser de Comandos no Cliente

O cliente quebra o input do usuário em 2 partes: o nome do comando propriamente dito e o parâmetro passado. Assim, ele pode avaliar se o parâmetro faz sentido, dependendo do comando, e tomar a ação necessária enviando ou recebendo as mensagens adequadamente.

6 Resolução de Comandos no Servidor

O servidor recebe cada comando enviado pelo cliente da maneira como foi escrito pelo usuário. Da mesma forma como feito pelo cliente, ele separa o nome do comando do parâmetro e toma as ações adequadas. Na maioria dos comandos ele executa uma *system call* e retorna a resposta adequada ao cliente, conforme os diagramas de sequência previamente mostrados.

7 Outros Exemplos

7.1 Transferência de Imagem

Realizou-se a transferência de um arquivo *.png* com sucesso, como mostrado na Figura 16. A Figura 17 mostra o arquivo transferido da imagem (que era apenas um *printscreeen* do terminal) sendo aberto normalmente.

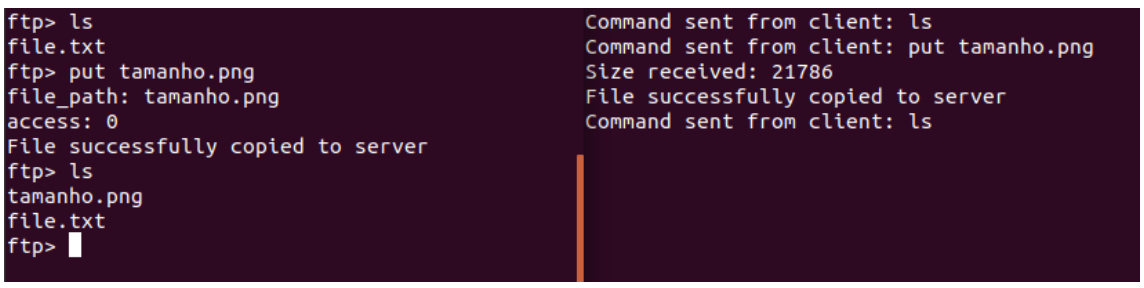


Figura 16: Exemplo de transferência de imagem

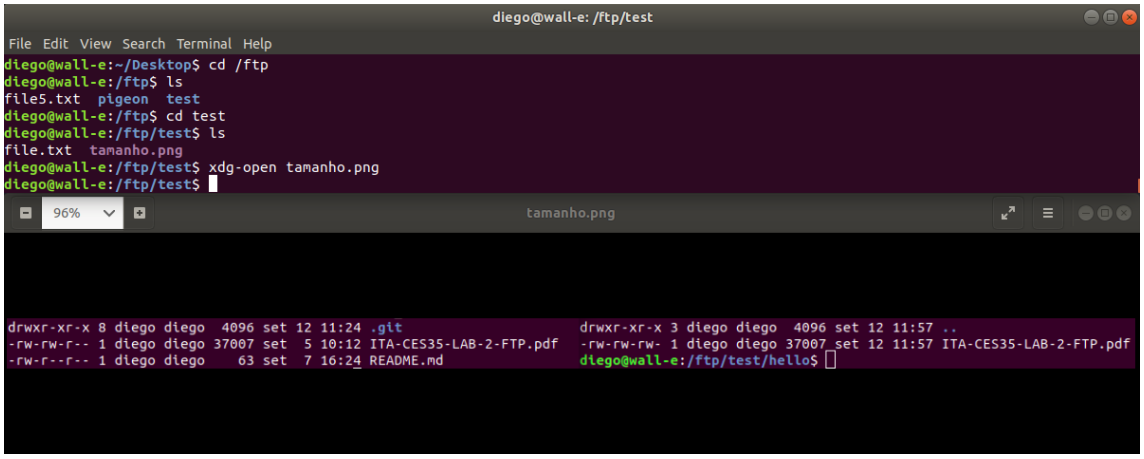


Figura 17: Imagem como foi recebida do lado do servidor

7.2 Outra Transferência de Imagem

Em seguida tentou-se fazer a transferência de uma imagem *png* obtida da internet e realizou-se o mesmo procedimento anterior. Infelizmente não foi possível transferir esse outro arquivo, conforme visto na Figura 18.

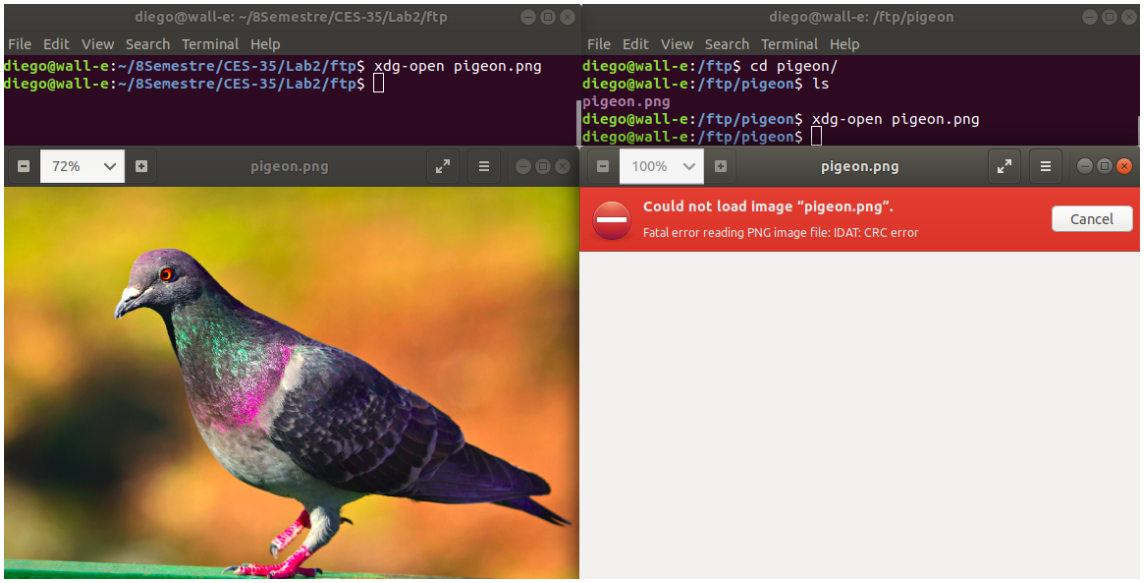


Figura 18: Tentativa de transferir outra imagem

Inicialmente considerou-se a possibilidade de ser o *encoding*, porém o arquivo enviado no exemplo anterior também era *png* e foi possível enviá-lo. Talvez haja algum problema na transferência de arquivos muito grandes, visto que o tamanho do arquivo anterior é de apenas 21 kB e o outro é de 820 kB.

8 Conclusão

Foi muito interessante desenvolver um serviço utilizando sockets e trabalhando com envio de mensagens. Apesar disso, foi um laboratório muito extenso e trabalhoso em especial porque a linguagem escolhida foi C. Infelizmente alguns arquivos não-txt não foram transferidos corretamente, como algumas imagens *png*, *jpg* e arquivos *pdf*. Isso provavelmente se deve a algum problema na transferência de arquivos muito grandes.