
*Instituto Federal de Minas Gerais*

*Bacharelado em Ciência da Computação*

# Relatório da Aplicação *FTP*

*Autores:*
Lucas de Oliveira Souza
Barbosa e Jean Claudio
Teixeira

*Docente:*
Danielle Costa

Disciplina:

*Redes de Computadores I*

Junho 20, 2023

# 1 Introdução

Este trabalho, trata-se de uma implementação de uma Aplicação FTP (*File Transfer Protocol*). Tal aplicação deve no final, realizar a transferência de arquivos entre cliente e servidor. Para isso, utiliza-se dois soquetes, uma para os dados e o outro para o controle.

# 2 Materiais usados

Para a implementação desta aplicação, foi utilizada a linguagem Java e o ambiente utilizado para tal, foi o *Visual Studio Code*. Ademais, para realização de consultas sobre o protocolo FTP, utilizou-se neste, o material disponibilizado pelo *site datatracker*, por (POSTEL; REYNOLDS, 1985).

# 3 Desenvolvimento

## 3.1 Lista de comandos

Abaixo tem-se a lista dos comandos disponíveis nessa aplicação:

1. *LIST*;
2. *RETR (RETRIEVE)*;
3. *STOR (STORE)*;
4. *CWD (CHANGE WORKING DIRECTORY)*;
5. *PWD (PRINT WORKING DIRECTORY)*;
6. *CDUP (CHANGE TO PARENT DIRECTORY)*.

## 3.2 Servidor

Em relação as bibliotecas utilizadas no servidor, tem-se as seguintes:

```
1    import java.io.*;
2    import java.net.ServerSocket;
3    import java.net.Socket;
4    import java.util.Arrays;
5    import java.util.List;
```

Listing 1: Bibliotecas usadas.

Abaixo, ilustra-se a função principal (*main*) do servidor, que caso necessário, faça uma alteração nas variáveis da Porta de Controle (*controlPort*) e da Porta de dados (*dataPort*). E, claro, definir o diretório padrão para os arquivos (*DEFAULT_DIRECTORY*).

```
public class FTPServer {
    public static final String DEFAULT_DIRECTORY = "\\
       directory\\"; // Defina o diretorio padrao para os
       arquivos

    public static void main(String[] args) {
        int controlPort = 12384;
        int dataPort = 8888;
        try {
            ServerSocket controlSocket = new ServerSocket(
               controlPort);
            System.out.println("Servidor FTP iniciado.");
            System.out.println("Aguardando conexoes na porta
               " + controlPort + "...");

            while (true) {
                Socket clientSocket = controlSocket.accept();
                Thread clientThread = new Thread(new
                   ClientHandler(clientSocket, dataPort));
                clientThread.start();
            }
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

Listing 2: Função principal do servidor.

Abaixo, ilustra-se a seção que irá rodar até o usuário entrar com o comando "*QUIT*", que acarretará no fechamento do soquete de controle e no encerramento da conexão.

```
class ClientHandler implements Runnable {
    private Socket controlSocket;
    private int dataPort;
    public String currentDirectory;

    public ClientHandler(Socket controlSocket, int dataPort)
       {
        this.controlSocket = controlSocket;
        this.dataPort = dataPort;
        this.currentDirectory = FTPServer.DEFAULT_DIRECTORY;
    }

    public void run() {
        try {
            String clientAddress = controlSocket.
               getInetAddress().getHostAddress();
            System.out.println("Conexao estabelecida com " +
               clientAddress);
            File dirAtual = new File("D:\\directory\\");
            BufferedReader controlReader = new BufferedReader
               (new InputStreamReader(controlSocket.
               getInputStream()));
            PrintWriter controlWriter = new PrintWriter(
```

```
                controlSocket.getOutputStream(), true);

            boolean connectionClose = false;
            while (!connectionClose) {
              String command = controlReader.readLine();
              String[] commandParts = command.split(" ");
              String line;

              if (commandParts[0].equals("RETR")) {
          //Logica para o comando RETR;
              }else if (commandParts[0].equals("LIST")) {
          //Logica para o comando LIST;
              }else if(commandParts[0].equals("STOR")) {
          //Logica para o comando STOR;
              }else if (commandParts[0].equals("PWD")) {
          //Logica para o comando PWD;
              }else if (commandParts[0].equals("CWD")) {
          //Logica para o comando CWD;
              }else if (commandParts[0].equals("CDUP")) {
        //Logica para o comando CDUP;
              }else if (commandParts[0].equals("QUIT"))
                 connectionClose = true;
              else controlWriter.println("ERROR: Comando
                 invalido");
            }
            controlSocket.close();
        } catch (IOException e) {e.printStackTrace();}
    }
}
```

Listing 3: Implementação do Servidor.

Abaixo, ilustra-se a lógica realizada no servidor para a execução do comando *RETR (Retrieve)*.

```
        String filename = commandParts[1];
        File file = new File(dirAtual+"\\"+ filename);
        if (file.exists()) {
            controlWriter.println("OK");
            // Cria um soquete de dados para o cliente
            ServerSocket dataSocket = new ServerSocket(
               dataPort);
            controlWriter.println(dataPort);
            Socket dataConnection = dataSocket.accept();
            BufferedOutputStream dataWriter = new
               BufferedOutputStream(dataConnection.
               getOutputStream());
            FileInputStream fileReader = new FileInputStream(
               file);
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = fileReader.read(buffer)) !=
               -1)
```

```
                dataWriter.write(buffer, 0, bytesRead);
            fileReader.close();
            dataWriter.flush();
            dataWriter.close();
            dataConnection.close();
            dataSocket.close();
            System.out.println("Arquivo '" + filename + "'
                enviado para " + clientAddress);
        } else controlWriter.println("ERROR: Arquivo nao
            encontrado");
```

Listing 4: Lógica do comando RETR.

Abaixo, ilustra-se a lógica realizada no servidor para a execução do comando *LIST*.

```
        controlWriter.println("OK");
        // Cria um soquete de dados para o cliente
        ServerSocket dataSocket = new ServerSocket(dataPort);
        controlWriter.println(dataPort);
        Socket dataConnection = dataSocket.accept();
        PrintWriter dataWriter = new PrintWriter(
            dataConnection.getOutputStream(), true);
        File[] files = dirAtual.listFiles();
        for (File file : files)
            dataWriter.println(file.getName());
        dataWriter.close();
        dataConnection.close();
        dataSocket.close();
        System.out.println("Lista de arquivos enviada para "
            + clientAddress);
```

Listing 5: Lógica do comando LIST.

Abaixo, ilustra-se a lógica realizada no servidor para a execução do comando *STOR (STORE)*.

```
        controlWriter.println("OK");
        File file = new File(commandParts[2] + "\\" +
            commandParts[1]);
        System.out.println(file.toString());
        // Cria um soquete de dados para o cliente
        ServerSocket dataSocket = new ServerSocket(dataPort);
        controlWriter.println(dataPort);
        Socket dataConnection = dataSocket.accept();
        BufferedInputStream dataReader = new
            BufferedInputStream(dataConnection.getInputStream
            ());
        FileOutputStream fileWriter = new FileOutputStream(
            file);
        byte[] buffer = new byte[1024];
        int bytesRead;
        while ((bytesRead = dataReader.read(buffer)) != -1)
            fileWriter.write(buffer, 0, bytesRead);
        fileWriter.close();
```

```
15          dataReader.close();
16          dataConnection.close();
17          dataSocket.close();
18          System.out.println("Arquivo '" + commandParts[1] + "'
                recebido e salvo em: " + file.getAbsolutePath());
```

Listing 6: Lógica do comando STOR.

Abaixo, ilustra-se a lógica realizada no servidor para a execução do comando *PWD (PRINT WORKING DIRECTORY)*.

```
1           controlWriter.println("OK");
2           String currentDirectory = dirAtual.getAbsolutePath();
3           controlWriter.println(currentDirectory);
```

Listing 7: Lógica do comando PWD.

Abaixo, ilustra-se a lógica realizada no servidor para a execução do comando *CWD (CHANGE WORKING DIRECTORY)*.

```
1           String[] arquivos = dirAtual.list();
2           List listaArquivo = Arrays.asList(arquivos);
3           if (listaArquivo.contains(commandParts[1])) {
4               File novoDir = new File(dirAtual + "\\" +
                    commandParts[1]);
5               String caminhoNovo = novoDir.getAbsolutePath();
6               if (novoDir.isDirectory())
7                   dirAtual = novoDir;
8           }
9           controlWriter.println("OK");
10          controlWriter.println(dirAtual);
```

Listing 8: Lógica do comando CWD.

Abaixo, ilustra-se a lógica realizada no servidor para a execução do comando *CDUP (CHANGE TO PARENT DIRECTORY)*.

```
1           File novoDir = new File(dirAtual.getParent());
2           dirAtual = novoDir;
3           controlWriter.println("OK");
4           controlWriter.println(dirAtual);
```

Listing 9: Lógica do comando CDUP.

## 3.3 Cliente

Em relação as bibliotecas utilizadas no cliente, tem-se as seguintes:

```
1       import java.io.*;
2       import java.net.Socket;
3       import java.net.UnknownHostException;
4       import java.util.Scanner;
```

Listing 10: Bibliotecas usadas no cliente.

Abaixo, ilustra-se a função principal (*main*) do cliente, que caso necessário, faça uma alteração nas variáveis da Porta de Controle (*controlPort*), da Porta de dados (*dataPort*) e do endereço do servidor (*serverAddress*).

```
1     public static void main(String[] args) throws
         UnknownHostException, IOException {
2     String serverAddress = "localhost";
3     int controlPort = 12384;
4     int dataPort = 8888;
5     Socket controlSocket = new Socket(serverAddress,
         controlPort);
6
7     String userInput;
8     do {
9          System.out.print("ftp> ");
10         Scanner ler = new Scanner(System.in);
11         userInput =ler.nextLine();
12         String[] comando = userInput.split(" ");
13
14         switch (comando[0]) {
15         case "LIST":
16             listFiles(controlSocket,serverAddress,
                controlPort, dataPort);
17             break;
18         case "RETR":
19             retrieveFile(controlSocket,serverAddress,
                controlPort, dataPort, comando[1],
                DEFAULT_DOWNLOAD_DIRECTORY);
20             break;
21         case "STOR":
22             storeFile(controlSocket,serverAddress,
                controlPort, dataPort, comando[1], comando[2])
                ;
23             break;
24         case "PWD":
25             printWorkingDirectory(controlSocket);
26             break;
27         case "CWD":
28             changeWorkingDirectory(controlSocket, comando[1])
                ;
29             break;
30         case "CDUP":
31             changetoParentDirectory(controlSocket);
32             break;
33         case "QUIT":
34             System.out.println("Goodbye...");
35             break;
36         default:
37             throw new IllegalArgumentException("Comando " +
                userInput+" nao reconhecido.");
38         }
39    } while (!userInput.equals("QUIT"));
40    try {controlSocket.close();}
41    catch (IOException e) {e.printStackTrace();}
```

```
}
```

Listing 11: Função principal do cliente.

Abaixo, ilustra-se a função presente no cliente que realiza a lógica para a execução do comando *RETR (Retrieve)*.

```
    public static void retrieveFile(Socket controlSocket,
       String serverAddress, int controlPort, int dataPort,
       String filename, String downloadDirectory) {
    try {
        PrintWriter controlWriter = new PrintWriter(
           controlSocket.getOutputStream(), true);
        BufferedReader controlReader = new BufferedReader(new
            InputStreamReader(controlSocket.getInputStream())
           );

        controlWriter.println("RETR " + filename);
        String response = controlReader.readLine();

        if (response.equals("OK")) {
            int serverDataPort = Integer.parseInt(
               controlReader.readLine());
            Socket dataSocket = new Socket(serverAddress,
               serverDataPort);
            BufferedInputStream dataReader = new
               BufferedInputStream(dataSocket.getInputStream
               ());
            FileOutputStream fileWriter = new
               FileOutputStream(downloadDirectory + filename)
               ;
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = dataReader.read(buffer)) !=
               -1)
                fileWriter.write(buffer, 0, bytesRead);
            fileWriter.close();
            dataReader.close();
            dataSocket.close();
            System.out.println("Arquivo '" + filename + "'
               recebido e salvo em: " + downloadDirectory);
        } else System.out.println("Erro: " + response);
    } catch (IOException e) {e.printStackTrace();}
}
```

Listing 12: Função executável do comando RETR.

Abaixo, ilustra-se a função presente no cliente que realiza a lógica para a execução do comando *LIST*.

```
    public static void listFiles(Socket controlSocket,String
       serverAddress, int controlPort, int dataPort) {
    try {
        PrintWriter controlWriter = new PrintWriter(
           controlSocket.getOutputStream(), true);
```

```
        BufferedReader controlReader = new BufferedReader(new
            InputStreamReader(controlSocket.getInputStream())
           );

        controlWriter.println("LIST");
        String response = controlReader.readLine();
        if (response.equals("OK")) {
            int serverDataPort = Integer.parseInt(
               controlReader.readLine());
            Socket dataSocket = new Socket(serverAddress,
               serverDataPort);
            BufferedReader dataReader = new BufferedReader(
               new InputStreamReader(dataSocket.
               getInputStream()));

            String line;
            while ((line = dataReader.readLine()) != null)
                System.out.println(line);
            dataReader.close();
            dataSocket.close();
            System.out.println("Lista de arquivos recebida
               com sucesso.");
        } else System.out.println("Erro: " + response);
    } catch (IOException e) {e.printStackTrace();}
}
```

Listing 13: Função executável do comando LIST.

Abaixo, ilustra-se a função presente no cliente que realiza a lógica para a execução do comando *PWD (PRINT WORKING DIRECTORY)*.

```
  public static void printWorkingDirectory(Socket
     controlSocket) {
    try {
        PrintWriter controlWriter = new PrintWriter(
           controlSocket.getOutputStream(), true);
        BufferedReader controlReader = new BufferedReader(new
            InputStreamReader(controlSocket.getInputStream())
           );

        controlWriter.println("PWD");
        String response = controlReader.readLine();
        if (response.equals("OK")) {
            String currentDirectory = controlReader.readLine
               ();
            System.out.println("Diretorio atual: " +
               currentDirectory);
        } else System.out.println("Erro: " + response);
    } catch (IOException e) {e.printStackTrace();}
}
```

Listing 14: Função executável do comando PWD.

Abaixo, ilustra-se a função presente no cliente que realiza a lógica para a execução do comando *CWD (CHANGE WORKING DIRECTORY)*.

```
1   public static void changeWorkingDirectory(Socket
       controlSocket, String comando) {
2       try {
3           PrintWriter controlWriter = new PrintWriter(
               controlSocket.getOutputStream(), true);
4           BufferedReader controlReader = new BufferedReader
               (new InputStreamReader(controlSocket.
               getInputStream()));
5
6           controlWriter.println("CWD "+comando);
7           String response = controlReader.readLine();
8           if (response.equals("OK")) {
9               String currentDirectory = controlReader.
                   readLine();
10              System.out.println("Diretorio atual: " +
                   currentDirectory);
11          } else System.out.println("Erro: " + response);
12      } catch (IOException e) {e.printStackTrace();}
13  }
```

Listing 15: Função executável do comando CWD.

Abaixo, ilustra-se a função presente no cliente que realiza a lógica para a execução do comando *CDUP (CHANGE TO PARENT DIRECTORY)*.

```
1   public static void changetoParentDirectory(Socket
       controlSocket) {
2     try {
3         PrintWriter controlWriter = new PrintWriter(
              controlSocket.getOutputStream(), true);
4         BufferedReader controlReader = new BufferedReader(
              new InputStreamReader(controlSocket.
              getInputStream()));
5
6         controlWriter.println("CDUP");
7         String response = controlReader.readLine();
8         if (response.equals("OK")) {
9             String currentDirectory = controlReader.
                  readLine();
10            System.out.println("Diretorio Alterado com
                  sucesso \nDiretorio atual: " +
                  currentDirectory);
11        } else System.out.println("Erro: " + response);
12    } catch (IOException e) {e.printStackTrace();}
13  }
```

Listing 16: Função executável do comando CDUP.

Abaixo, ilustra-se a função presente no cliente que realiza a lógica para a execução do comando *STOR (STORE)*.

```
1     public static void storeFile(Socket controlSocket,
          String serverAddress, int controlPort, int dataPort,
          String filename, String directory) {
2         try {
3             PrintWriter controlWriter = new PrintWriter(
                  controlSocket.getOutputStream(), true);
4             BufferedReader controlReader = new BufferedReader
                  (new InputStreamReader(controlSocket.
                  getInputStream()));
5
6             controlWriter.println("STOR " + filename + " " +
                  directory);
7             String response = controlReader.readLine();
8             if (response.equals("OK")) {
9                 // Cria um soquete de dados para enviar o
                      arquivo
10              int serverPort = Integer.parseInt(controlReader
                    .readLine());
11                Socket dataSocket = new Socket(serverAddress,
                      serverPort);
12                BufferedOutputStream dataWriter = new
                      BufferedOutputStream(dataSocket.
                      getOutputStream());
13                File arquivo = new File(
                      DEFAULT_DOWNLOAD_DIRECTORY+filename);
14                FileInputStream fileReader = new
                      FileInputStream(arquivo);
15                byte[] buffer = new byte[1024];
16                int bytesRead;
17
18                while ((bytesRead = fileReader.read(buffer))
                      != -1)
19                    dataWriter.write(buffer, 0, bytesRead);
20                fileReader.close();
21                dataWriter.flush();
22                dataWriter.close();
23                dataSocket.close();
24                System.out.println("Arquivo '" + filename + "
                      ' enviado para o servidor");
25            } else System.out.println("Erro: " + response);
26
27        } catch (IOException e) {e.printStackTrace();}
28    }
```

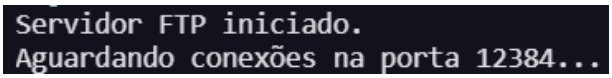
Listing 17: Função executável do comando STOR.

# 4 Conclusão

Com toda essa implementação, tem-se uma aplicação FTP com uma arquitetura cliente/servidor básica para a transferência de arquivos em rede. E, sua saída no terminal é a

seguinte:

1. Saída do servidor:




2. Saída do cliente: (obs.: "*<command>*" é apenas um exemplo de onde o usuáio tem de inserir o comando.


# 5 Referências

POSTEL, J.; REYNOLDS, J.. FILE TRANSFER PROTOCOL (FTP). 1985. Disponível em: *datatracker*. Acesso em: 15 jun. 2023.