
UNIVERSIDADE FEDERAL DE PERNAMBUCO

Centro de Informática
UFPE

**Universidade Federal de Pernambuco**
**Centro de Informática - CIn**

# Relatório do Projeto: Programação utilizando sockets

Redes de Computadores
Docentes: Kelvin Lopes Dias e Maria Katarine de Santana Barbosa

Caio Buarque de Gusmão, Diego Henrique Vilaca Calixto, José Gabriel Santos Nascimento e Julia Arnaud de Melo Fragoso

Recife, 27 de Outubro de 2022

# 1. Introdução

O trabalho desenvolvido pelo nosso grupo teve como objetivo criar um servidor de autenticação e um servidor web a partir da implementação de sockets, para um entendimento mais prático acerca dos protocolos do modelo TCP/IP envolvidos na disciplina de Redes de Computadores.

Primeiramente, para o planejamento do projeto foram utilizadas ferramentas como o Notion, Google Docs e GitHub, onde o conteúdo estudado foi organizado e compartilhado entre os membros do grupo. Determinamos reuniões semanais, com objetivos distribuídos para cada membro, onde eram mostrados os avanços de cada um e a implementação do projeto como um todo era discutida.

Nas primeiras semanas de trabalho, nosso grupo focou no desenvolvimento do servidor de autenticação, pesquisando sobre o tema e elaborando os primeiros códigos. Assim, o processo foi evoluindo de maneira constante, desenvolvendo um protocolo da camada de aplicação que utiliza tanto o algoritmo de criptografia diffie-hellman, quanto o protocolo da camada de transporte TCP, sendo capaz de se conectar com múltiplos clientes simultaneamente. Junto a essa autoridade certificadora também foi desenvolvido um servidor web, que implementa o método GET do protocolo HTTP, sendo capaz de retornar diversos tipos de arquivos.

# 2. Detalhamento da implementação

Antes de nos aprofundarmos detalhadamente na implementação do projeto, é importante que sejam mencionadas as bibliotecas que foram usadas para dar suporte ao código. Elas estão listadas abaixo.

### Sockets

Esta é uma biblioteca que permite a utilização dos sockets, que são essenciais para o envio de dados através da rede, baseado nos protocolos da camada de transporte. Toda a comunicação do projeto entre servidor e cliente foi feita baseada nessa API.

### Cryptocode

Esta é uma biblioteca responsável por realizar a criptografia de dados de uma maneira simples, transformando ela através do recebimento de uma chave e uma string, ela utiliza o algoritmo de criptografia AES-GCM garantindo segurança e integridade para as informações comunicadas.

### uuid

Esta é uma biblioteca responsável por gerar códigos chamados de Identificadores Universalmente Únicos, através dela é criado um código de 128 bits representado por 32 dígitos hexadecimais, que irá servir como identificador para cada cliente que se comunicará com o servidor.

### Pickle
Pickle é um módulo nativo do python que permite a serialização e deserialização da estrutura de objetos. É utilizada no nosso projeto para salvar e carregar o banco de dados dos clientes em um arquivo, podendo continuar de onde parou mesmo com a interrupção da execução do código.

### Threading
Threading é uma biblioteca do Python que tem como funcionalidade a implementação do conceito de thread de processos no código. Thread pode ser definido como pedaços de um processo que podem ser escalonados independentemente, permitindo para o código do projeto a possibilidade de atender diversos clientes paralelamente.

### Cryptography.fernet
Cryptography.fernet é uma biblioteca de criptografia que utiliza um algoritmo simétrico, ou seja, ela é capaz de criptografar arquivos utilizando uma chave única, para além de criptografar, descriptografar-los, tornando eles em um formato inacessível quando criptografado, funcionando para arquivos além de .txt, como arquivos .png, através do método de leitura/escrita de arquivo permitido pelo python.

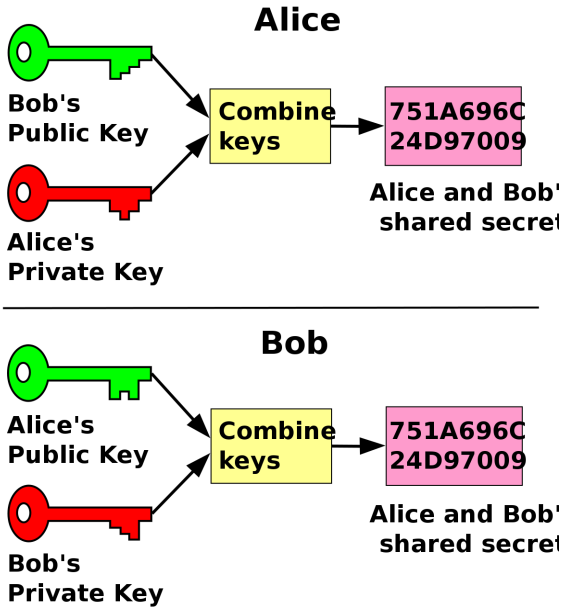
### Random
Biblioteca responsável por gerar números aleatórios. Foi usada para ajudar a gerar a chave privada do nosso servidor.

### Rsa
Esta biblioteca foi utilizada para desenvolver a assinatura digital, ela suporta criptografia e descriptografia, assinatura, verificação de assinatura e, além disso, é capaz de criar as chaves de acordo com a função PKCS#1. Assim, conseguimos gerar as chaves que foram usadas para assinar o GET e verificar essa assinatura.

### Email.utils, Datetime e Time
Ambas as três foram usadas para informar a data no cabeçalho do 200 Ok, 400 Bad Request, 403 Forbidden e 404 Not Found. Tanto na formatação quanto na data e horário corretos.

Uma vez que as bibliotecas foram explicadas, podemos discutir mais a fundo sobre a implementação. Para a primeira parte do projeto, referente ao servidor de autenticação, desenvolvemos o algoritmo de criptografia baseado na troca de chaves do algoritmo de Diffie-Hellman onde o cliente e o servidor compartilharão informações através do Handshake de maneira pública e através delas irão gerar uma chave secreta. Essas informações contemplam duas chaves públicas: a chave P, sendo um número primo e a chave G, sendo uma raiz primitiva de P, ambas escolhidas previamente; E duas chaves privadas, uma do servidor e outra do cliente, escolhidas a cada execução como um número aleatório entre 1 e 64.

Isso evita que um terceiro, que tenha acesso a essa troca de mensagens, seja capaz de conseguir a chave secreta, chave essa que será usada mais para frente para as criptografias. A forma que foi utilizada o algoritmo de Diffie-Hellman na comunicação entre sockets foi inspirada pelas etapas do protocolo TLS (Cliente Hello, Server Hello, Change Cipher, Handshake Finished), realizando as etapas de comunicação mas utilizando o cálculo de Diffie-Hellman através dos números modulares compartilhados entre servidor-cliente.



Nas imagens abaixo é possível ver como se inicia essa comunicação na implementação foi feita:

**Cliente:**

```python
def Handshake(mClientSocket, A_ChavePrivClient):

    mensagem = "CLIENT HELLO   "  # ta fazendo a primeira requisição pro servidor
    mClientSocket.send(mensagem.encode())  # Ta notificando pro servidor que esse é o client hello, ou seja, ta pedindo as chaves publicas

    while True:

        req = mClientSocket.recv(2048)
        req = req.decode()  # recebe a resposta do servidor, se tudo der certo tem que receber o server hello
        req = req[:15]

        if req == "SERVER HELLO   ":
            P_ChavePubServ = mClientSocket.recv(2048)  # recebendo as duas chaves publicas
            P_ChavePubServ = int(P_ChavePubServ.decode())

            G_ChavePubServ = mClientSocket.recv(2048)
            G_ChavePubServ = int(G_ChavePubServ.decode())  # decodificando para int

            # cipher é um numero pré criptografia que utliza as chaves publicas e privadas para fazer um
            # segredo que vai ser compartilhado entre o cliente e servidor
            # vai ser usado como parametro para calcular a chave secreta
            # tambem é chamado de chave modular
            X_cipherCliente = int(pow(G_ChavePubServ, A_ChavePrivClient, P_ChavePubServ))

            # transformando para string para mandar para o servidor
            # a função encode só aceita string como parametro para codificar
            # por isso tou transformando em string antes de mandar
            X_cipherCliente = str(X_cipherCliente)

            mensagem = "CHANGE CIPHER  "  # cabeçalho para o servidor entender o que ta acontecendo (troca dos segredos compartilhados)
            mClientSocket.send(mensagem.encode())

            mClientSocket.send(X_cipherCliente.encode())  # mandando o cipher
            Y_cipherServidor = mClientSocket.recv(2048)  # recebendo o cipher
            Y_cipherServidor = int(Y_cipherServidor.decode())  # decodificando direto pra inteiro para poder calcular dps
```

```python
    if req == "RSA CHANGE KEY ":  # faz a troca de chaves da biblioteca de assinatura digital
        resp = "RSA CHANGE KEY "
        mClientSocket.send(resp.encode())  # reponde, notificando para o servidor que irá ocorrera a troca de chaves

        (rsa_chave_pub_cliente, rsa_chave_priv_cliente) = rsa.newkeys(2048)

        rsa_chave_pub_cliente = rsa_chave_pub_cliente.save_pkcs1(format="DER")  # serializa a chave publica do cliente para bytes
        mClientSocket.send(rsa_chave_pub_cliente)  # envia a chave publica do cliente para o servidor


    if req == "HANDSHAKE FIN  ":
        chave_secreta_cliente = int(pow(Y_cipherServidor, A_ChavePrivClient, P_ChavePubServ))  # calculo da chave secreta

        mClientSocket.send(req.encode())  # alertanado para o servidor que ja possui a chave secreta

        return str(chave_secreta_cliente), rsa_chave_priv_cliente
```

**Servidor:**

```python
# Essa função faz o handshake para obter a chave de criptografia
def Handshake(mClientSocket, P_ChavePubServ, G_ChavePubServ, B_ChavePrivServ):

    while True:  # while para manter a conexão com o cliente enquanto for necessário
        # Recebendo os dados do Cliente:
        data = mClientSocket.recv(2048)  # recebe a primeira requisição do cliente
        req = data.decode()  # a resposta é recebida em bytes, por isso é preciso transformar bytes em string para podermos entender
        req = req[:15]

        if req == "CLIENT HELLO   ":  # respondendo o client hello enviado pelo cliente, ele quer as chaves publicas

            resp = "SERVER HELLO   "  # da a resposta para o cliente saber que esse é o server hello e o conteudo é as chaves publicas
            mClientSocket.send(resp.encode())  # envia a requisicao

            mClientSocket.send(P_ChavePubServ.encode())  # envia as chaves publicas
            mClientSocket.send(G_ChavePubServ.encode())

        if req == "CHANGE CIPHER  ":  # ocorre a troca do segredo compartilhado (cipher), necessário para a calcular a chave secreta
            X_cipherCliente = mClientSocket.recv(2048)  # ta recebendo o segredo compartilhado do cliente
            X_cipherCliente = X_cipherCliente.decode()
            X_cipherCliente = int(X_cipherCliente)  # transformando para int para poder calcular depois

            G_ChavePubServ = int(G_ChavePubServ)  # transformando para int para poder realizar o calculo
            P_ChavePubServ = int(P_ChavePubServ)

            Y_cipherServidor = int(pow(G_ChavePubServ, B_ChavePrivServ, P_ChavePubServ))  # calculo do cipher do servidor
            Y_cipherServidor = str(Y_cipherServidor)  # transformando para string para poder mandar para o cliente
            mClientSocket.send(Y_cipherServidor.encode())  # mandando o cipher para o cliente

            resp = "RSA CHANGE KEY "  # notificando que a troca de informações foi concluida e que ja pode ser feito o calculo da chave secreta
            mClientSocket.send(resp.encode())
```

```python
    if req == "RSA CHANGE KEY ":  # ocorre a troca de chaves da biblioteca de assinatura digital

        rsa_chave_pub_cliente = mClientSocket.recv(2048)  # recebe a chave publica do cliente
        rsa_chave_pub_cliente = rsa.PublicKey.load_pkcs1(rsa_chave_pub_cliente, format="DER")  # serializa a chave publica do cliente de bytes para
        # o objeto especifico da biblioteca

        # serialização é uma maneira de transformar bytes em objetos de python, ou seja, de bytes para variável útil para nós
        resp = "HANDSHAKE FIN  "
        mClientSocket.send(resp.encode())

    if req == "HANDSHAKE FIN  ":  # recebe do cliente que ele ja fez a chave secreta e também o servidor pode fazer a chave secreta dele

        chave_secreta_servidor = int(pow(X_cipherCliente, B_ChavePrivServ, P_ChavePubServ))  # calculo da chave secreta

        # o return ja acaba com a função e retorna a chave secreta
        # a variável manter_conexão no while não precisa por causa desse return
        # a chave secreta é transformada em string por causa da função de criptografia que exige ela em string para criptografar
        return str(chave_secreta_servidor), rsa_chave_pub_cliente
```

Primeiramente, ocorre a comunicação do cliente intitulada de “Cliente Hello” que pede ao servidor as suas chaves públicas, este que responde notificando como “Server Hello” suas chaves públicas. Ao receber as chaves públicas o cliente requisita, com o nome “Change Cipher” a chave modular do servidor , chave essa que é calculada a partir das chaves públicas e da chave privada e é utilizada como parâmetro para o cálculo da chave secreta futuramente. Nesse contexto, o cliente calcula a sua chave modular e envia para o servidor, o servidor recebe a chave modular do cliente, calcula a sua própria chave modular e a envia para o cliente. Agora ambas as partes têm as informações necessárias para o cálculo da chave secreta, mas antes é preciso fazer a troca de chaves responsáveis pela assinatura digital. A assinatura digital é feita pela biblioteca RSA e para executá-la o cliente cria um par de chaves, uma pública e outra privada, a pública deve ser compartilhada com o servidor a fim que verifique a assinatura futuramente, nesse sentido, essa troca acontece no handshake no processo chamado “RSA Change Key”, onde o cliente gera essas chaves e envia sua chave pública para o servidor. Por fim, o servidor notifica o final do handshake

com o “Handshake Fin” onde é calculada tanto a chave secreta do servidor quanto a chave secreta do cliente.

Agora que a conexão foi estabelecida com o servidor, o cliente poderá realizar o seu pedido, entretanto ele deverá seguir os 4 pilares de segurança de redes (confidencialidade, integridade, autenticidade e disponibilidade). O projeto conseguiu atingir esses 4 pilares e será visto como que eles foram implementados separadamente.

## Thread

Uma característica solicitada para o servidor foi a capacidade de atender múltiplos clientes enquanto em execução, para solucionar esse problema nós aplicamos o conceito de thread para o nosso servidor. Na computação, thread pode ser definido como um subprocesso que pode ser escalonado para CPU individualmente, ou seja, é um pedaço do processo que executa como um programa independente. Quando implementado, foi utilizado a biblioteca threading que fez da função *HandleRequest* uma thread. Nesse sentido, toda vez que um cliente se conecta com o nosso servidor, uma thread dessa função é criada e cada cliente é processado individualmente.

```
# Colocar o servidor para escutar as solicitações de conexão
mSocketServer.listen()
while True:
    # Este loop foi colocado para que o servidor conseguisse se conectar com vários cliente;
    clientSocket, clientAddr = mSocketServer.accept()
    print(f'\nO servidor aceitou a conexão do Cliente: {clientAddr}')

    # Criação de múltiplas threads para que o servidor consiga responder mais de um cliente por vez.
    Thread(target=HandleRequest, args=(
        clientSocket,
        clientAddr,
        P_ChavePubServ,
        G_ChavePubServ,
        B_ChavePrivServ,
        banco_de_dados)).start()
```

No código vemos que um while é implementado para aceitar clientes continuamente e a Thread com o parâmetro “target = HandleRequest” passando a função a ser executada e “args” com os argumentos.

## Handle Request

Agora que entendemos a utilização da thread na implementação do servidor vamos detalhar o que a função *HandleRequest* faz. Ela é a principal função do código, é a partir dela que outras funções são chamadas.

```python
# Essa função é uma thread que lida com a comunicação via sockets de cada cliente
def HandleRequest(mClientSocket, mClientAddr, P_ChavePubServ, G_ChavePubServ, B_ChavePrivServ, banco_de_dados):
    # inicialmente vamos ver se o cliente ja se comunicou antes com o servidor atraves de um indentificador unico
    # se ele não possuir um indentificador ou não for achado um vai ser criado um novo
    c1, existe_chave = Acharindentificador(mClientSocket, banco_de_dados, mClientAddr)
    # essa função retorna c1, que é a estrutura de dados do cliente contendo informações sobre ele
    # aqui por exemplo foi printado o endereço apenas para testar
    # e também retorna existe chave que verifica se o cliente ja possui as chaves

    print(f"Cliente {clientAddr} com indentificador : {c1.indentificador}")
    # caso não exista chaves é necessário cria-las através do handshake
    if not existe_chave:
        # em seguida é feito o handshake para adquirir a chave secreta
        chave_secreta_servidor, rsa_chave_pub_cliente = Handshake(mClientSocket, P_ChavePubServ, G_ChavePubServ, B_ChavePrivServ)
        # é salvo as chaves de acordo com o cliente
        c1.chave_secreta = chave_secreta_servidor
        c1.rsa_chave_secreta = rsa_chave_pub_cliente

    # A próxima etapa consiste em atender as requisições GET do cliente
    GetHandler(mClientSocket, chave_secreta_servidor, rsa_chave_pub_cliente)
```

Primeiramente é chamado a função *AcharIndentificador* responsável por verificar se um cliente já possui um identificador único e se já existem chaves vinculadas àquele cliente, se não existir é necessário chamar o handshake para criar essas chaves e vinculá-las ao cliente específico. Por fim a função chama a *GetHandler* que vai lidar com as requisições GET

## Cliente

Para melhor organizar os dados de cada cliente foi criada uma estrutura de dados chamada Cliente, nela é armazenado o identificador, endereço, chave secreta para criptografia e chave secreta para assinatura digital.

```python
class Cliente:
    def __init__(self, indentificador, endereço):
        self.indentificador = indentificador
        self.endereço = endereço
        self.chave_secreta = None
        self.rsa_chave_secreta = None
```

## Banco de Dados

O banco de dados está organizado em um dicionário onde a chave é o identificador e o objeto cliente é seu par.

```
{'3fb6d834-5596-11ed-b4d3-00e04c05b934': <DadosCliente.Cliente object at 0x00000129FB7C79A0>,
 '46c8f9c8-5596-11ed-a4da-00e04c05b934': <DadosCliente.Cliente object at 0x00000129FBB2C040>}
```

Ele é armazenado em um arquivo externo chamado “banco_de_dados.txt” e é inicializado ao executar o servidor e atualizado toda vez que um cliente se conecta. Para conseguir armazená-lo em um arquivo foi utilizada a biblioteca Pickle que faz a serialização de objetos para arquivos txt, dessa maneira é possível armazenar e restaurar objetos de uma maneira mais simples.

## Identificador

Para cada cliente ser identificado e referenciado no servidor foi necessário criar um identificador único. Para isso foi utilizada a biblioteca UUID que gera um identificador de 128 bits com uma mínima probabilidade de colisão entre identificadores.

```python
def Acharindentificador(mClientSocket, banco_de_dados, mClientAddr):
    existe_chave = False
    indentificador = mClientSocket.recv(2048)  # recebe do cliente um indentificador ou "None" caso não tenha indentificador
    indentificador = indentificador.decode()

    if indentificador != "None":  # se existir indentificador (indentificador não for None) procurar no banco de dados
        # tratamento de erro para caso ele não esteja no banco de dados
        try:
            c1 = banco_de_dados[indentificador]  # testa para ver se ta no banco de dados
            resp = "ID OK"  # se tiver responde para o cliente dizendo que o indentificador esta ok
            mClientSocket.send(resp.encode())
            existe_chave = True

        except KeyError:  # se der erro (o indentificador não estiver no banco de dados) comunicar o cliente e criar um novo
            resp = "NOT FOUND"
            mClientSocket.send(resp.encode())
            indentificador, c1 = NovoIndentificador(banco_de_dados, mClientAddr)
            mClientSocket.send(indentificador.encode())

    # se não existir indentificador criar um e enviar para o cliente
    else:
        indentificador, c1 = NovoIndentificador(banco_de_dados, mClientAddr)
        resp = "NEW ID"
        mClientSocket.send(resp.encode())
        mClientSocket.send(indentificador.encode())

    return c1, existe_chave  # retorna a estrutura de dados cliente para o código
```

A função achar identificador chamada anteriormente, tem como argumentos o socket do cliente, um banco de dados e o endereço do cliente. A execução ocorre com o recebimento de um identificador do cliente onde é verificado se já existe no banco de dados, a verificação ocorre com um tratamento de erro ao tentar acessar um valor de um dicionário com sua chave correspondente. Caso exista o identificador a função retorna o cliente presente no banco de dados, caso o identificador enviado pelo cliente não seja encontrado ou não exista (None) é criado um novo identificador a partir de outra função.

```python
# essa função é reponsável por criar um novo indentificador e salvar como novo cliente no banco de dados
def NovoIndentificador(banco_de_dados, mClientAddr):
    # é utilizado a função uuid1 da biblioteca uuid para criar um conjunto de caracteres que vão servir
    # para como indentificador único do cliente
    indentificador = uuid1()
    # a biblioteca cria o identificador numa estrutura de dados dela e a gente passa para string para melhor manipulação
    indentificador = str(indentificador)

    # criação da estrutura de dados cliente
    c1 = Cliente(indentificador, mClientAddr)  # é passado como parametro o indentificador e o endenreço do seu acesso
    # setattr(banco_de_dados, indentificador, c1)
    banco_de_dados[indentificador] = c1  # salvando no banco de dados o cliente com o seu id

    return indentificador, c1  # retornamos o id e o cliente
```

Essa função gera um identificador e cria um novo cliente na estrutura do nosso código passando como parâmetro o id criado e o endereço do cliente, por fim, é adicionado no banco de dados e o cliente em formato de objeto, junto com o identificador, é retornado

## Get

O get é a principal razão pela qual o servidor funciona: transmitir arquivos. Para isso, a ideia da transferência é simples, o cliente faz uma requisição de um arquivo e o servidor manda o arquivo pedido para o cliente. Mas para garantir a segurança foram implementadas algumas ferramentas.

O requisição get se inicia no cliente que após receber do usuário o arquivo desejado vai para a função *GET*, nela, essa requisição é criptografada e assinada e enviada para o servidor, que faz toda a verificação para garantir a criptografia e que quem enviou foi de fato o cliente, através da assinatura digital, além disso, o cliente faz uma troca de chaves de uma biblioteca específica para criptografia de arquivos, com o servidor.

```
# função que lida com as requisições get
def GET(mClientSocket, req, rsa_chave_priv_cliente, chave_secreta_cliente):


    # assinatura digital
    req = cryptocode.encrypt(req, chave_secreta_cliente)  # criptografia da requisição do get
    req = req.encode()  # transformando em bytes para mandar pro servidor
    mClientSocket.send(req)

    req_assinado = rsa.sign(req, rsa_chave_priv_cliente, 'SHA-512')  # assinatura digital
    mClientSocket.send(req_assinado)  # envio da assinatura digital para o servidor

    #troca de chave de criptografia de arquivo

    with open("filekey.key", "wb") as key_file:
        crypt_keybits = mClientSocket.recv(2048)
        crypt_key_decode = crypt_keybits.decode()
        crypt_key = cryptocode.decrypt(crypt_key_decode, chave_secreta_cliente)

        crypt_key = crypt_key.encode()
        key_file.write(crypt_key)

    with open('filekey.key', 'rb') as filekey:  # Lendo a chave recebida
        key = filekey.read()

    usable_key = Fernet(key)
```

Em seguida, o servidor faz a verificação de sintaxe do pedido, observando os formatos suportados.

```
lista_de_formatos = ["txt", "html", "htm", "css", "js", "png", "jpg", "svg", "pdf"]
formato = req.split(".")
formato = formato[-1]

if formato not in lista_de_formatos:
    Erro_400(mClientSocket, chave_secreta_servidor)
    return
```

E, por fim, verifica se o arquivo existe tentando abri-lo, retornando um erro caso não exista, após isso, o arquivo é criptografado utilizando a biblioteca Fernet e enviado para o cliente.

```
try:
    with open(req, 'rb') as file:  # Leitura do arquivo para varíavel
        original = file.read()

    Msg_200(mClientSocket, chave_secreta_servidor)

    encrypted = usable_key.encrypt(original)  # Criptografando a varíavel que contem o arquivo

    with open(req, 'wb') as encrypted_file:  # Escrita do Arquvio Criptografado
        encrypted_file.write(encrypted)

    with open(req, 'rb') as file:  # Seleciona o arquvio pedido
        file = file.read() # Leitura das linhas do arquivo
        clientSocket.send(file)  # Envio das linhas do arquivo para o cliente

    with open(req, 'wb') as final_file:
        final_file.write(original)

except FileNotFoundError:
    Erro_404(mClientSocket, chave_secreta_servidor, req)
    return
```

O cliente, por sua vez, recebe o cabeçalho, apresentando o status da mensagem recebida, e o próprio arquivo que é salvo em seu diretório.

```
cabecalho = mClientSocket.recv(2048)
cabecalho = cabecalho.decode()
cabecalho = cryptocode.decrypt(cabecalho, chave_secreta_cliente)
print(cabecalho)

if cabecalho[9:12] != "200":
    return


with open(req, "wb") as file:  # Inicio da criação do arquivo pedido
    # While permanecer enquanto ouver linhas para serem escritas
        # Recebimento das linhas do arquivo 1 milhão de bits pois as linhas são imprevisíveis
    data = mClientSocket.recv(1000000)
    # Escrita das linhas do arquivo na pasta do Cliente
    file.write(data)

with open(req, "rb") as crip_file:
    crypted = crip_file.read()

decrypt = usable_key.decrypt(crypted)

with open(req, 'wb') as final_file:
    final_file.write(decrypt)
```

Observação: caso o status da mensagem não seja o código 200 OK o recebimento do arquivo não é efetuado e o GET finalizado.

**Criptografia de string**

Para o pedido do cliente não ser capaz de ser acessado, foi utilizado uma criptografia AES-GCM através da biblioteca Cryptocode, criptografando a string que

representa o pedido antes dele ser enviado, onde só é possível realizar a descriptografia tendo em posse a chave secreta, que somente o cliente e o servidor tem acesso.

O passo a passo é o seguinte:

1 - Cliente escreve o pedido no terminal.
2 - A mensagem do cliente é criptografada usando a chave secreta.
3 - A mensagem agora criptografada é transformada em bits e enviada ao servidor.
4 - O servidor recebe essa mensagem em bits e transforma ela para string.
5 - Para finalizar o Servidor descriptografa a mensagem, tendo em posse pôr fim do pedido feito pelo cliente

Na implementação:

Cliente:

```
# função que lida com as requisições get
def GET(mClientSocket, req):
    req = cryptocode.encrypt(req, chave_secreta_cliente)  # criptografia da requisição do ge
    req = req.encode()  # transformando em bytes para mandar pro servidor
    mClientSocket.send(req)
```

Servidor:

```
req = req.decode()  # da decode nos bytes para por na função de descriptografia em str
req = cryptocode.decrypt(req, chave_secreta_servidor)  # decriptografado utilziando a
```

Detalhe a ser observado, “req” é o equivalente a requisição do cliente.

**Criptografia de arquivos**

Outra parte da implementação que foi necessário seguir o requisito de confidencialidade foi na entrega do arquivo requerido feito pelo servidor, e o recebimento pelo cliente. Nesse contexto, o arquivo tem que está em um estado que não possa ser acessado por quem não seja o servidor ou o cliente, e para realizar essa ação foi utilizado a biblioteca Cryptography.Fernet, que é capaz de tornar um arquivo ilegível através de sua criptografia.

A criptografia funciona da seguinte forma:

1 - Como a chave de criptografia de arquivo é enviada ao cliente.

1.1 - O servidor gera uma chave aleatória utilizando a biblioteca.

```
key = Fernet.generate_key()
```

1.2 - Essa chave precisa ser transformada em “usável” para aplicar na criptografia

```
usable_key = Fernet(key)
```

1.3 - Como ela se encontra em formato de bits quando criada é necessário dar decode, e utilizar a biblioteca Cryptocode para enviar ao cliente de maneira segura.

```
decode_key = key.decode()
crypt_key = cryptocode.encrypt(decode_key, chave_secreta_servidor)
clientSocket.send(crypt_key.encode())
```

2 - Como o servidor usa a chave para criptografar o arquivo e enviar ao cliente

2.1 - Primeiramente o arquivo desejado é passado para uma variável através da leitura do mesmo.

```
with open(req, 'rb') as file:  # Leitura do arquivo para varíavel
    original = file.read()
```

2.2 - O arquivo na variável é criptografado utilizando a chave e a biblioteca, e sobrescrevendo o arquivo original.

```
encrypted = usable_key.encrypt(original)  # Criptografando a variavel que contem o arquivo

with open(req, 'wb') as encrypted_file:  # Escrita do Arquvio Criptografado
    encrypted_file.write(encrypted)
```

2.3 - Agora com o arquivo criptografado o servidor envia para o cliente os bits correspondentes ao arquivo.

```
with open(req, 'rb') as file:  # Seleciona o arquvio pedido
    file = file.read() # Leitura das linhas do arquivo
    clientSocket.send(file)  # Envio das linhas do arquivo para o cliente
```

2.4 - Por fim, o arquivo criptografado pelo servidor agora será reescrito utilizando sua versão original para permanecer a sua integridade.

```
with open(req, 'wb') as final_file:
    final_file.write(original)
```

3 - Veremos agora como funciona com o cliente.

3.1 - O cliente cria um arquivo para ser escrito, onde ficará a chave da Fernet para ser usada posteriormente, antes disso ele converte os bits para string, e descriptografar usando a biblioteca Cryptocode.

3.2 - O arquivo da chave da Fernet é repassado para uma variável para criar uma variável

```
with open('filekey.key', 'rb') as filekey:  # Lendo a chave recebid
    key = filekey.read()

usable_key = Fernet(key)
```

3.3 - Agora o arquivo criptografado é recebido e escrito na pasta do Cliente.

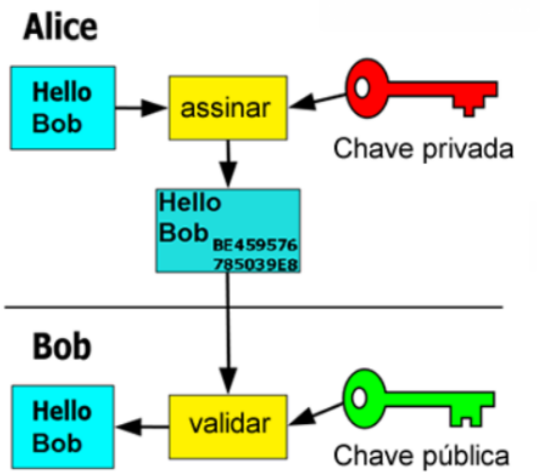
```
with open(req, "wb") as file:  # Inicio da criação do arquivo pedido

    # Recebimento do arquivo 1 milhão de bits pois são imprevisíveis
    data = mClientSocket.recv(1000000)

    # Escrita das linhas do arquivo na pasta do Cliente
    file.write(data)
```

3.4 - Agora o arquivo é passado para uma variável, descriptografado pela biblioteca e escrito sobre o antigo(criptografado).

```
with open(req, "rb") as crip_file:
    crypted = crip_file.read()

decrypt = usable_key.decrypt(crypted)

with open(req, 'wb') as final_file:
    final_file.write(decrypt)
```

**Assinatura Digital**

Para fazer a assinatura digital, que garante autenticidade, utilizamos a biblioteca rsa, que gerou chaves públicas e privadas. Nesse sentido, nosso código seguiu a lógica de assinar a mensagem GET do cliente, fazendo um hash com a sua chave privada, e verificar essa assinatura do lado do servidor, com a chave pública, que é compartilhada entre eles. Assim, o servidor tem conhecimento de quem está fazendo aquela requisição e pode avaliar se essa pessoa está autorizada a ter acesso ao documento requisitado. A imagem a seguir pode ajudar na compreensão de como esse método foi implementado.




Abaixo é possível ver a função, do lado do cliente, que cria as chaves e compartilha a chave pública com o servidor, utilizando a biblioteca RSA.

```python
if req == "RSA CHANGE KEY ":  # faz a troca de chaves da biblioteca de assinatura digital
    resp = "RSA CHANGE KEY "
    mClientSocket.send(resp.encode())  # reponde, notificando para o servidor que irá ocorrera a troca de chaves

    (rsa_chave_pub_cliente, rsa_chave_priv_cliente) = rsa.newkeys(2048)

    rsa_chave_pub_cliente = rsa_chave_pub_cliente.save_pkcs1(format="DER")  # serializa a chave publica do cliente para bytes
    mClientSocket.send(rsa_chave_pub_cliente)  # envia a chave publica do cliente para o servidor
```

Na próxima imagem observa-se que o GET é assinado através de um hash que usa a chave privada do cliente.

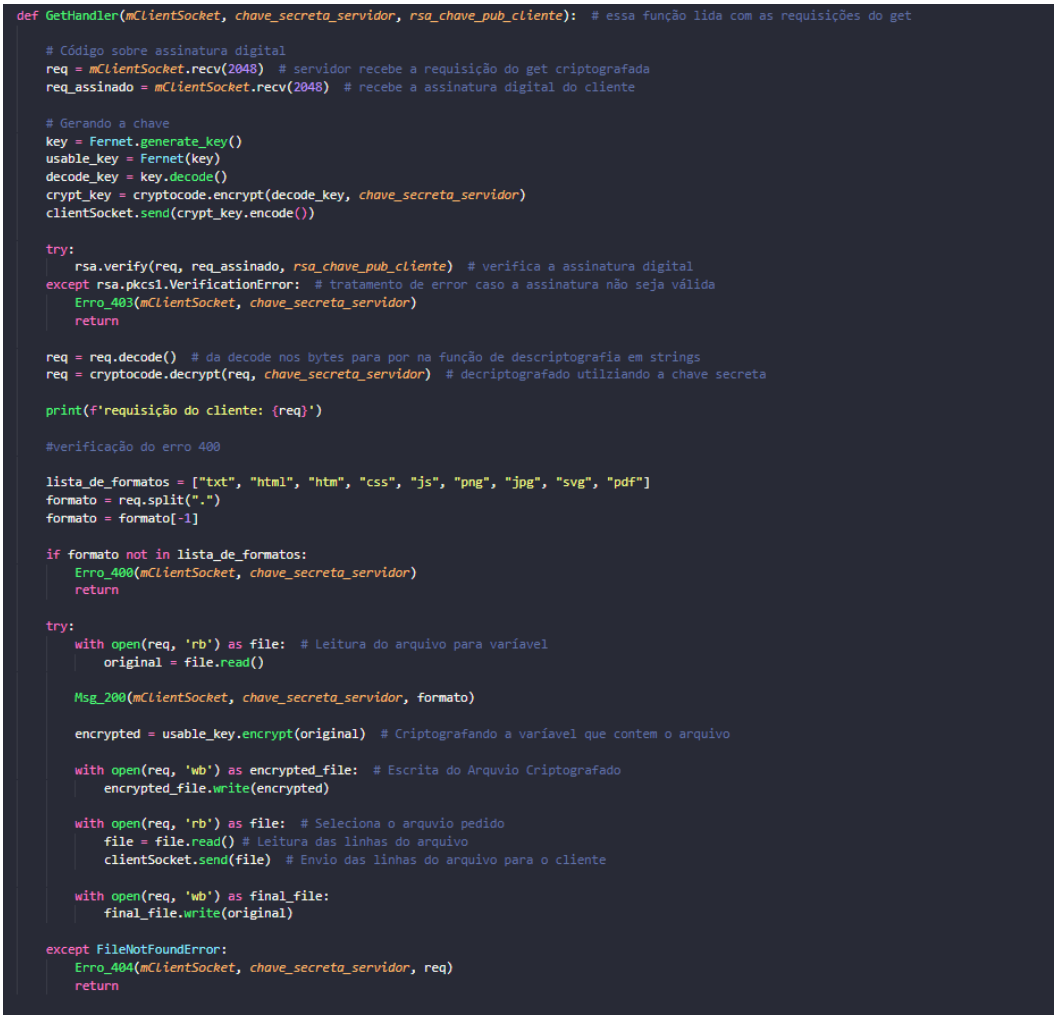
```python
def GET(mClientSocket, req, rsa_chave_priv_cliente):

    # assinatura digital
    req = cryptocode.encrypt(req, chave_secreta_cliente)  # criptografia da requisição do get
    req = req.encode()  # transformando em bytes para mandar pro servidor
    mClientSocket.send(req)

    req_assinado = rsa.sign(req, rsa_chave_priv_cliente, 'SHA-512')  # assinatura digital
    mClientSocket.send(req_assinado)  # envio da assinatura digital para o servidor

    req = req.decode()
    req = cryptocode.decrypt(req, chave_secreta_cliente)
```

Por fim, podemos ver que, do lado do servidor, a assinatura digital é verificada com a função verify, autenticando a mensagem.

```python
try:
    rsa.verify(req, req_assinado, rsa_chave_pub_cliente)  # verifica a assinatura digital
except rsa.pkcs1.VerificationError:  # tratamento de error caso a assinatura não seja válida
    print("verificação falhou")  # verificação falhou lidar com isso no tratamento de error
    # error 403
```

## Códigos de Erro

É requisito para o projeto as respostas 200 OK, 400 Bad Request, 403 Forbidden e 404 Not Found, e, foram implementadas da seguinte forma:



```python
def GetHandler(mClientSocket, chave_secreta_servidor, rsa_chave_pub_cliente):  # essa função lida com as requisições do get

    # Código sobre assinatura digital
    req = mClientSocket.recv(2048)  # servidor recebe a requisição do get criptografada
    req_assinado = mClientSocket.recv(2048)  # recebe a assinatura digital do cliente

    # Gerando a chave
    key = Fernet.generate_key()
    usable_key = Fernet(key)
    decode_key = key.decode()
    crypt_key = cryptocode.encrypt(decode_key, chave_secreta_servidor)
    clientSocket.send(crypt_key.encode())

    try:
        rsa.verify(req, req_assinado, rsa_chave_pub_cliente)  # verifica a assinatura digital
    except rsa.pkcs1.VerificationError:  # tratamento de error caso a assinatura não seja válida
        Erro_403(mClientSocket, chave_secreta_servidor)
        return

    req = req.decode()  # da decode nos bytes para por na função de descriptografia em strings
    req = cryptocode.decrypt(req, chave_secreta_servidor)  # decriptografado utilziando a chave secreta

    print(f'requisição do cliente: {req}')

    #verificação do erro 400

    lista_de_formatos = ["txt", "html", "htm", "css", "js", "png", "jpg", "svg", "pdf"]
    formato = req.split(".")
    formato = formato[-1]

    if formato not in lista_de_formatos:
        Erro_400(mClientSocket, chave_secreta_servidor)
        return

    try:
        with open(req, 'rb') as file:  # Leitura do arquivo para varíavel
            original = file.read()

        Msg_200(mClientSocket, chave_secreta_servidor, formato)

        encrypted = usable_key.encrypt(original)  # Criptografando a varíavel que contem o arquivo

        with open(req, 'wb') as encrypted_file:  # Escrita do Arquvio Criptografado
            encrypted_file.write(encrypted)

        with open(req, 'rb') as file:  # Seleciona o arquvio pedido
            file = file.read() # Leitura das linhas do arquivo
            clientSocket.send(file)  # Envio das linhas do arquivo para o cliente

        with open(req, 'wb') as final_file:
            final_file.write(original)

    except FileNotFoundError:
        Erro_404(mClientSocket, chave_secreta_servidor, req)
        return
```

(figura 1 deste tópico - função getHandler do lado servidor)

```python
# função que lida com as requisições get
def GET(mClientSocket, req, rsa_chave_priv_cliente, chave_secreta_cliente):


    # assinatura digital
    req = cryptocode.encrypt(req, chave_secreta_cliente)  # criptografia da requisição do get
    req = req.encode()  # transformando em bytes para mandar pro servidor
    mClientSocket.send(req)

    req_assinado = rsa.sign(req, rsa_chave_priv_cliente, 'SHA-512')  # assinatura digital
    mClientSocket.send(req_assinado)  # envio da assinatura digital para o servidor



    #troca de chave de criptografia de arquivo

    with open("filekey.key", "wb") as key_file:
        crypt_keybits = mClientSocket.recv(2048)
        crypt_key_decode = crypt_keybits.decode()
        crypt_key = cryptocode.decrypt(crypt_key_decode, chave_secreta_cliente)

        crypt_key = crypt_key.encode()
        key_file.write(crypt_key)

    with open('filekey.key', 'rb') as filekey:  # Lendo a chave recebida
        key = filekey.read()

    usable_key = Fernet(key)


    req = req.decode()
    req = cryptocode.decrypt(req, chave_secreta_cliente)

    # inicio do cabeçalho

    cabecalho = mClientSocket.recv(2048)
    cabecalho = cabecalho.decode()
    cabecalho = cryptocode.decrypt(cabecalho, chave_secreta_cliente)
    print(cabecalho)

    if cabecalho[9:12] != "200":
        return
```

(Figura 2 deste tópico - função get do lado do cliente)

1- **200 OK**

Vai ocorrer quando a requisição for bem sucedida e o objeto será enviado. Se tudo ocorrer da maneira desejada (o cliente tem permissão, fez a requisição da maneira correta e existe o arquivo que ele está procurando), o código entra na função “Msg_200(mClientSocket, chave_secreta_servidor, formato)” (figura 1 deste tópico - função getHandler do lado servidor).

```python
def Msg_200(mClientSocket, chave_secreta_servidor, formato):
    now = datetime.now()
    mStamp = mktime(now.timetuple())

    resposta = ''
    resposta += 'HTTP/1.1 200 OK\r\n'
    resposta += f'Date: {formatdate(timeval=mStamp, localtime=False, usegmt=True)}\r\n'
    resposta += 'Server: CIn UFPE/0.0.0.1 (Ubuntu)\r\n'
    # resposta += f'Content-Length: '
    resposta += f'Content-Type: {formato}\r\n' #alterar para o formato do arquivo enviado
    resposta += '\r\n'

    resposta = cryptocode.encrypt(resposta, chave_secreta_servidor)
    resposta = resposta.encode()
    mClientSocket.send(resposta)
```

Nessa função vai se fazer o cabeçalho da mensagem (usa-se o parâmetro “formato” para se colocar no cabeçalho o formato do arquivo (.png, .txt …)) que vai ser criptografado (com o uso do argumento *chave_secreta_servidor*) e enviado ao cliente (com o uso do argumento *mClientSocket)*.

2- **400 Bad Request**

Acontecerá quando a requisição não for entendida pelo servidor, no nosso caso, o tipo de arquivo foi escrito errado. O tipo de arquivo será analisado na lista “lista_de_formatos”, se o formato não estiver na lista, entrará na função “Erro_400(*mClientSocket*, *chave_secreta_servidor*)” (figura 1 deste tópico - função getHandler do lado servidor).

```
def Erro_400(mClientSocket, chave_secreta_servidor):
    now = datetime.now()
    mStamp = mktime(now.timetuple())

    # header
    resposta = ''
    resposta += 'HTTP/1.1 400 Bad Request\r\n'
    resposta += f'Date: {formatdate(timeval=mStamp, localtime=False, usegmt=True)}\r\n'
    resposta += 'Server: Projeto/127.0.0.1 (Ubuntu)\r\n'
    # resposta += f'Content-Length: '
    resposta += 'Content-Type: text/html\r\n'
    resposta += '\r\n'

    # mensagem
    html = ''
    html += '<html>'
    html += '<head>'
    html += '<title>Redes de Computadores - CIn/UFPE</title>'
    html += '<meta charset="UTF-8">'
    html += '</head>'
    html += '<body>'
    html += '<h1>400 Bad Resquest</h1>'
    html += '<h2>your client has issued a malformed or illegal request.</h2>'
    html += '</body>'
    html += '</html>'

    resposta += html

    resposta += html
    resposta = cryptocode.encrypt(resposta, chave_secreta_servidor)
    resposta = resposta.encode()
    mClientSocket.send(resposta)
```

Nessa função irá se fazer o cabeçalho do erro e sua mensagem html, esses serão reunidos em uma única variável que será criptografada (com o uso do argumento *chave_secreta_servidor*) e enviada do servidor para o cliente (com o uso do argumento *mClientSocket)*, onde será descriptografado e printado no terminal do cliente. Essa última parte pode ser vista na figura 2 deste tópico, a partir do “# inicio do cabeçalho”.

3**- 403 Forbidden**

Quando um cliente não tem acesso ao servidor e/ou arquivo, no nosso caso, a assinatura digital não condizia com o que o servidor esperava. No nosso código é possível analisar que vai-se tentar verificar a assinatura digital (try), se não for possível vai cair no “except rsa.pkcs1.VerificationError:” e consequentemente no “Erro_403(*mClientSocket*, *chave_secreta_servidor*)” (figura 1 deste tópico - função getHandler do lado servidor).

```python
def Erro_403(mClientSocket, chave_secreta_servidor):
    now = datetime.now()
    mStamp = mktime(now.timetuple())

    # header
    resposta = ''
    resposta += 'HTTP/1.1 403 Forbidden\r\n'
    resposta += f'Date: {formatdate(timeval=mStamp, localtime=False, usegmt=True)}\r\n'
    resposta += 'Server: Projeto/127.0.0.1 (Ubuntu)\r\n'
    # resposta += f'Content-Length: '
    resposta += 'Content-Type: text/html\r\n'
    resposta += '\r\n'

    # mensagem
    html = ''
    html += '<html>'
    html += '<head>'
    html += '<title>Redes de Computadores - CIn/UFPE</title>'
    html += '<meta charset="UTF-8">'
    html += '</head>'
    html += '<body>'
    html += '<h1>forbidden</h1>'
    html += f'<h2>you dont have permission to acess this file on this server.</h2>'
    html += '</body>'
    html += '</html>'

    resposta += html
    resposta = cryptocode.encrypt(resposta, chave_secreta_servidor)
    resposta = resposta.encode()
    mClientSocket.send(resposta)
```

Nessa função irá se fazer o cabeçalho do erro e sua mensagem html, esses serão reunidos em uma única variável que será criptografada (com o uso do argumento *chave_secreta_servidor*) e enviada do servidor para o cliente (com o uso do argumento *mClientSocket)*, onde será descriptografado e printado no terminal do cliente. Essa última parte pode ser vista na figura 2 deste tópico, a partir do “# inicio do cabeçalho”.

4- **404 Not Found**

Ocorre quando um documento requisitado não está no servidor. Se o arquivo não for encontrado, vai cair no “except FileNotFoundError:” e, consequentemente, na função “Erro_404(*mClientSocket*, *chave_secreta_servidor*, req)” (figura 1 deste tópico - função getHandler do lado servidor).

```python
def Erro_404(mClientSocket, chave_secreta_servidor, req):
    now = datetime.now()
    mStamp = mktime(now.timetuple())

    # header
    resposta = ''
    resposta += 'HTTP/1.1 404 Not Found\r\n'
    resposta += f'Date: {formatdate(timeval=mStamp, localtime=False, usegmt=True)}\r\n'
    resposta += 'Server: Projeto/127.0.0.1 \r\n'
    # resposta += f'Content-Length: '
    resposta += 'Content-Type: text/html\r\n'
    resposta += '\r\n'

    # mensagem
    html = ''
    html += '<html>'
    html += '<head>'
    html += '<title>Redes de Computadores - CIn/UFPE</title>'
    html += '<meta charset="UTF-8">'
    html += '</head>'
    html += '<body>'
    html += '<h1>404 Error</h1>'
    html += '</a>'
    html += f'the file {req} was not found on the server'
    html += '</body>'
    html += '</html>'

    resposta += html
    resposta = cryptocode.encrypt(resposta, chave_secreta_servidor)
    resposta = resposta.encode()
    mClientSocket.send(resposta)
```

Nessa função irá se fazer o cabeçalho do erro e sua mensagem html (o argumento da função ‘req’ será usado para informar qual arquivo não foi encontrado), serão reunidos em uma única variável que será criptografada (com o uso do argumento *chave_secreta_servidor*) e enviada do servidor para o cliente (com o uso do argumento *mClientSocket)*, onde será descriptografado e printado no terminal do cliente. Essa última parte pode ser vista na figura 2 deste tópico, a partir do “# inicio do cabeçalho”.

→ Caso entra nos erros 400, 403 e 404 a conexão entre cliente e servidor será fechada, analisa-se isso com os “return” após saírem de suas funções de cabeçalho/mensagem (figura 1 deste tópico - função getHandler do lado servidor). É possível analisar isso também lado cliente (Figura 2 deste tópico - função get do lado do cliente), no último “if” da imagem, se o cabeçalho for diferente de 200 dá-se um “return”, fechando a conexão.

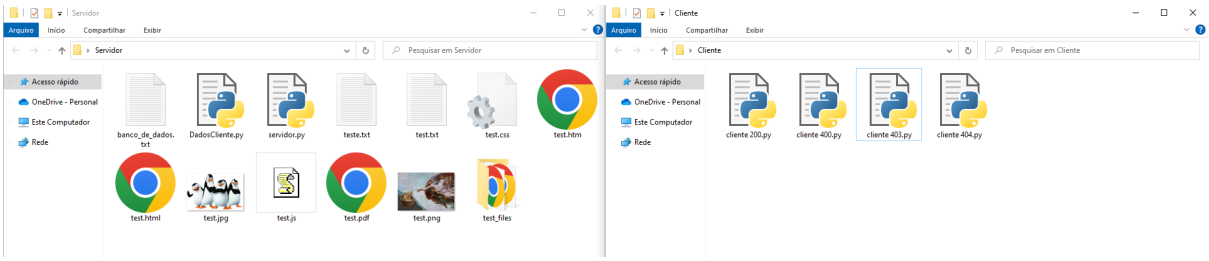
→ No fim do relatório estão os prints de como fica o terminal.

## Guia para o Usuário

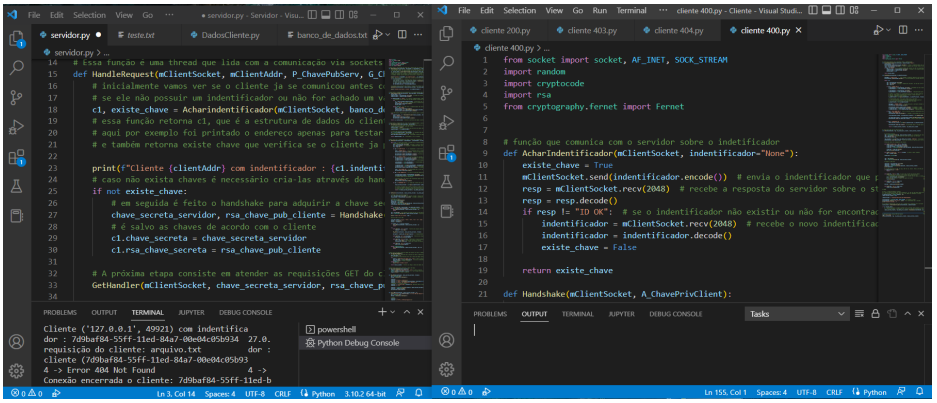
O conjunto Cliente-Servidor do nosso projeto consiste nos seguintes arquivos:

- Cliente 200.py
- Cliente 400.py
- Cliente 403.py
- Cliente 404.py
- servidor.py
- DadosCliente.py
- banco_de_dados.txt
- teste.txt
- test.html
- test.htm
- test.jpg
- test.png
- test.css
- test.js
- test.pdf

**Passo 1:**
Após o download é preciso que os arquivos sejam separados em duas pastas diferentes, uma contendo o servidor, com os arquivos: “servidor.py”, “DadosCliente.py”, “banco_de_dados.txt”, e os arquivos de teste para transferência. E outra contendo os clientes, contendo os arquivos: “Cliente 200.py”, “Cliente 400.py”, “Cliente 403.py”, “Cliente 404.py”.




**Passo 2:**
Ao utilizar a IDE de sua escolha é recomendável que abra as pastas em janelas separadas da IDE



**Passo 3:**
Baixar o python e todas as bibliotecas necessárias para a execução do código. Algumas já são nativas do python, portanto, é preciso a instalação das seguintes bibliotecas:
obs: o comando do terminal para instalar é “pip install *biblioteca*”

- cryptocode
- rsa
- cryptography

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
import random
import cryptocode
from DadosCliente import Cliente
import pickle
from uuid import uuid1
import rsa
from cryptography.fernet import Fernet
from email.utils import formatdate
from datetime import datetime
from time import mktime
```

**Passo 4:**
Executar o servidor. Ao aparecer a mensagem “Socket criado…” ele estará pronto para receber clientes

**Passo 5:**
Executar o cliente desejado.

**Observações:**
Importante exercitar a paciência com o código, devido ao prazo do projeto, ele não está devidamente otimizado. Nesse sentido, a execução do cliente normalmente demora entre 3 a 10 segundos. Caso o cliente apresente a mensagem “Conexão iniciada…” e demore mais do que o tempo comum, a execução apresentou algum problema desconhecido, para isso, o que funcionou para o nosso grupo é dar uma quebra de linha em qualquer espaço do código e executá-lo novamente.

## Pendências

Durante o desenvolvimento do projeto foi sempre buscado resolver um problema existente na permanência da conexão do cliente com o servidor. Nessa perspectiva, o problema que ocorria era que quando o cliente, dentro da Estrutura de Repetição, realizava o primeiro pedido ele não chegava, isso só acontecia ao forçar o encerramento do código do cliente, impedindo que os pedidos subsequentes fossem feitos. Porém, quando esse pedido era feito sem uma Estrutura de Repetição, com um único input e comando .send, ocorria tudo normalmente. Nesse sentido, tentamos de diversas formas resolver a situação, alterando a forma como a requisição era recebida no servidor, alternando a estrutura do get e alterando o método de envio, aplicando o método sleep, que pausava o código por um período de tempo, para caso fosse um problema de confusão nos envios simultâneos de bits impedindo o término do recebimento, mas nada resolvia a situação, e não conseguimos chegar a uma conclusão do problema, visto que se quer um erro aparecia no terminal, o código só permanecia em constante funcionamento, porém sem receber os arquivos.

## Dificuldades

Para tentar driblar as dificuldades, o grupo dividiu as tarefas com reuniões semanais, mas alguns problemas foram enfrentados mesmo assim. Inicialmente, o ponta pé inicial do trabalho foi um desafio a ser encarado, analisar as informações cedidas e entender como aplicá-las, foi necessário uma pesquisa para se aprofundar nos conceitos(Diffie-Hellman um exemplo), e saber onde eles se encaixavam no projeto, mas quando conseguimos superar essa barreira fomos capazes de avançar no servidor de autenticação com uma certa velocidade.

Quando avançados no funcionamento do servidor  de autenticação a busca pelo material que atingiria os pilares da segurança de redes foi sem dúvidas um desafio, pois não adiantava saber como eles funcionam, era necessário pensar em como eles se encaixavam no conjunto da relação servidor-cliente e mesmo que cada um tenha encontrado o meio para qualificar as etapas, foi necessário um trabalho árduo para juntar todas essas ferramentas e fazê-las funcionar. Além disso, alguns erros que não estávamos acostumados surgiam, por ser a nossa primeira experiência de um projeto envolvendo o uso de sockets, e trabalhar esses erros e entender o que eles significavam foi algo que atrasou o andamento do projeto, mas fomos capazes de contorná-los buscando até estratégias que fizessem com que esses erros não surgissem, otimizando o funcionamento do código.

Por fim, algo que apresentamos uma dificuldade foi conseguir realizar mudanças e testes no código, pois demandava uma grande organização por parte do grupo, onde diversas alterações no código, e isso precisava ser analisado e comparado, e visto a imensidão do código e por se tratar de mais um .py rodando foram utilizadas estratégias para lidar com essa problemática, como o uso de um Notion para deixar registrado cada etapa do código destrinchada para caso fosse necessário uma revisita, incluindo descrições além das presentes no próprio código para ajudar os outros membros do grupo a se manterem inteirados da situação.

## Conclusão

Com o desenvolvimento deste projeto tivemos que lidar com uma série de dificuldades, principalmente em relação à autoridade certificadora, onde tivemos que pesquisar sobre o tema, ler e estudar, para conseguir aplicar esses processos no código. Apesar disso, o grupo conseguiu contornar essas dificuldades e evoluir, assim, nosso aprendizado sobre a parte mais prática da disciplina e sobre as questões de segurança foi muito ampliado com esse trabalho, uma vez que tivemos que lidar com sockets, algoritmos de criptografia, assinatura digital, servidor web, threads, e várias outras ferramentas.

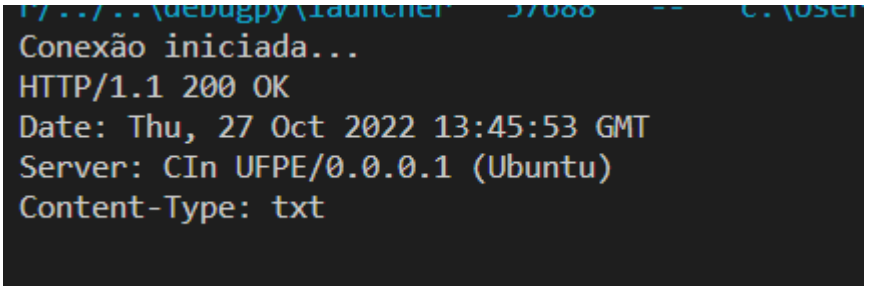
Além de tudo, o projeto nos proporcionou uma experiência de fornecer um conteúdo a prazo, semelhante a empresas de tecnologia, forçando o grupo a criar estratégias na produção do projeto visto a complexidade e nata de conhecimentos necessários para a sua realização, visto que se fosse somente um único indivíduo o realizando ele passaria por severas dificuldade.

A seguir será possível ver os prints do servidor em funcionamento através dos terminais:

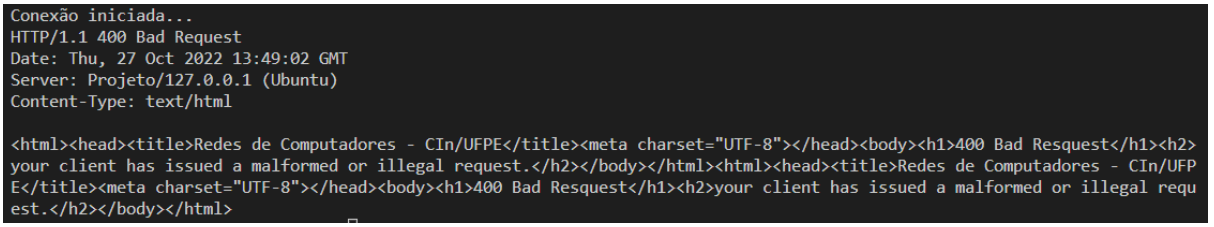
## Prints

### Clientes:

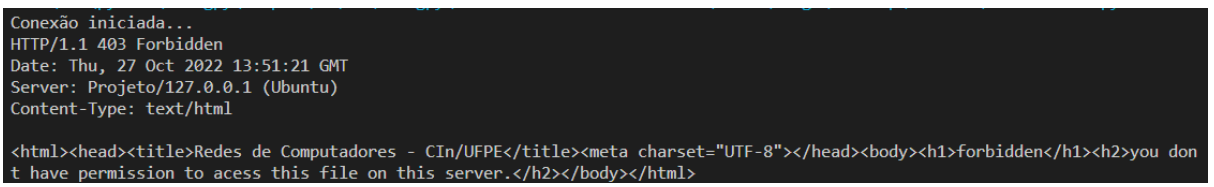
### Cliente 1




### Cliente 2




### Cliente 3



## Cliente 4

```
Conexão iniciada...
HTTP/1.1 404 Not Found
Date: Thu, 27 Oct 2022 13:53:39 GMT
Server: Projeto/127.0.0.1
Content-Type: text/html

<html><head><title>Redes de Computadores - CIn/UFPE</title><meta charset="UTF-8"></head><body><h1>404 Error</h1></a>the fil
e arquivo.txt was not found on the server</body></html>
```

## Servidor

```
Socket criado ...

O servidor aceitou a conexão do Cliente: ('127.0.0.1', 49904)
Cliente ('127.0.0.1', 49904) com indentificador : 75f09710-55ff-11ed-ad22-00e04c05b934
requisição do cliente: teste.txt
Conexão encerrada o cliente: 75f09710-55ff-11ed-ad22-00e04c05b934

O servidor aceitou a conexão do Cliente: ('127.0.0.1', 49912)
Cliente ('127.0.0.1', 49912) com indentificador : 79adedbe-55ff-11ed-b568-00e04c05b934
cliente (79adedbe-55ff-11ed-b568-00e04c05b934 -> Error 403 Forbidden
Conexão encerrada o cliente: 79adedbe-55ff-11ed-b568-00e04c05b934

O servidor aceitou a conexão do Cliente: ('127.0.0.1', 49921)
Cliente ('127.0.0.1', 49921) com indentificador : 7d9baf84-55ff-11ed-84a7-00e04c05b934
requisição do cliente: arquivo.txt
cliente (7d9baf84-55ff-11ed-84a7-00e04c05b934 -> Error 404 Not Found
Conexão encerrada o cliente: 7d9baf84-55ff-11ed-84a7-00e04c05b934

O servidor aceitou a conexão do Cliente: ('127.0.0.1', 49930)
Cliente ('127.0.0.1', 49930) com indentificador : 82261fbc-55ff-11ed-b068-00e04c05b934
requisição do cliente: teste.pmg
cliente (82261fbc-55ff-11ed-b068-00e04c05b934 -> Error 400 Bad Request
Conexão encerrada o cliente: 82261fbc-55ff-11ed-b068-00e04c05b934
```