# Relatório do projeto servidor-cliente utilizando sockets

**Angel Willyan Rodrigues de Lima[1], Gabriel Laroche Borba[2], João Lucas Vieira dos Santos[3], Luiz Guylherme Avelino Rodrigues[4]**

Centro de Informática – Universidade Federal de Pernambuco (UFPE)
[1]awrl@cin.ufpe.br,[2]glb@cin.ufpe.br,[3]jlvs@cin.ufpe.br,[4]lgar@cin.ufpe.br

***Resumo.*** *A lógica do sistema geral responsável pelo funcionamento da internet como conhecemos hoje, isto é, o conjunto de redes conectadas em prol do funcionamento mútuo, pode ser inescrutável quando paramos para olhar o modelo de rede apresentado nos dias atuais. Neste trabalho, observamos com o auxílio da linguagem de programação python, como pode ser feita a implementação de uma comunicação estável entre servidor e cliente, a qual possui a principal função de transporte de arquivos. Nosso objetivo foi observar como as comunicações mais básicas de redes podem facilitar nosso entendimento ao refletirmos sobre as aplicações mais complexas dos dias atuais.*

## 1. Introdução

Nosso projeto tem como foco a implementação de um servidor HTTP com sockets TCP, a fim de estabelecermos uma conexão estável e simularmos as aplicações reais do dia a dia. Para a realização do projeto, tivemos como base a linguagem de programação python 3, e, já que o intuito do projeto era demonstrar o funcionamento de uma rede HTTP do zero, utilizamos a programação por meio de sockets sem auxílio de qualquer biblioteca ou módulos pré-estabelecidos que implementam naturalmente a conexão.

Para que tenhamos uma base de conhecimentos amplos sobre todas as questões básicas da rede. Nosso projeto entra exatamente nesse quesito, precisamos entender como os processos mais simples são necessários na obtenção do conhecimento geral do modo como funciona o sistema geral de redes na atualidade.

Por isso, resolvemos começar do mais simples e fazer a implementação de um servidor-cliente, ou seja, um cliente que pode se comunicar com um servidor e vice-versa. A ideia geral do projeto é o cliente pedir arquivos e o servidor ser utilizado com o um "baú" que pode conter ou não os arquivos que o cliente peça. Além disso, deve ser implementada a criptografia de Diffie Hellman para protegermos todas as mensagens enviadas como forma de segurança e confiabilidade. Outrossim, um servidor que se preze deve conseguir atender vários clientes de uma vez, então, estipulamos uma meta de 5 clientes ao mesmo tempo.

## 2. Detalhamento da implementação

Aqui consta todos os detalhes da implementação do nosso projeto, desde quando o começamos até o final.

**2.1 Figuras, diagramas e algoritmos** de como o trabalho foi implementado.

**PROJETO 1 - CLIENTE**

Utilizamos 4 funções para o **cliente**.

```
def critogrtafiaAES(mensagem, senha): # funcao de criptografia
    mensagemCriptografada = cryptocode.encrypt(mensagem, f"{senha}")

    return mensagemCriptografada

def descriptografiaAES(mensagemCriptografada, senha): #funcao de descriptografia
    mensagemDescriptografada = cryptocode.decrypt(mensagemCriptografada, f"{senha}")

    return mensagemDescriptografada


def CriaGeradores(): #criou o gerador e primo comum aos dois
    geradorPrimo = random.randint(1,1000)
    gerador = random.randint(1,1000)

    while not primo(geradorPrimo):
        geradorPrimo = random.randint(1,1000)
    return geradorPrimo, gerador

def primo(n): #função para verificar se é primo
    for i in range(2,n):
        if n % i == 0:
            return False
    return True
```

As 2 primeiras funções são para criptografar e descriptografar as mensagens, respectivamente. Essas 2 funções estão presentes no servidor também. As 2 últimas funções são para criar um par ‘(número_aleatório, número_primo)’ que vão ser utilizadas no algoritmo de Diffie Hellman.

```
mClientSocket = socket(AF_INET, SOCK_STREAM)
mClientSocket.connect(('localhost', 1235))

geradorPrimo, gerador = CriaGeradores()
#Envia essas chaves para o servidor
chavesPrimoGerador = (f'chaves {geradorPrimo} {gerador}')
mClientSocket.send(chavesPrimoGerador.encode())

#O cliente recebe uma mensagem aprovando que o servidor
#recebeu o gerador e o primo
confimacao = mClientSocket.recv(2048)
req = confimacao.decode()
```

Após a definição das funções, temos o seguinte código que vai fazer a conexão TCP com o servidor, junto com o envio do par anteriormente citado para o servidor, seguido da resposta do servidor para comprovar que recebeu a chave.

```
if req == 'Chaves OK':
    #Cria uma chave confidencial do cliente
    chaveConfidencialCliente = random.randint(0,1000)
    #Executa o primeiro dieff hellman e envia para o servidor sua chave publica do cliente o rep1
    rep1 = (gerador**(chaveConfidencialCliente))%geradorPrimo
    mClientSocket.send(str(rep1).encode())

    #Recebe a chave publica do servidor
    data1 = mClientSocket.recv(2048)
    req1 = data1.decode()
    chavePublicaServidor = int(req1)
    print(f'Chave publica do servidor = {chavePublicaServidor}')

    #gera a chave compartilhada do cliente e servidor
    chaveCompartilhada = (chavePublicaServidor**chaveConfidencialCliente)%geradorPrimo
    print(f'chave compartilhada = {chaveCompartilhada}')

    #Envia para o servidor a chave compartilhada
    mClientSocket.send(str(chaveCompartilhada).encode())
    #Recebe a chave compartilhada do servidor
    data2 = mClientSocket.recv(2048)
    req2 = data2.decode()
```

Se o servidor retornar que as chaves anteriormente enviadas foram recebidas, o cliente vai prosseguir para a criação da chave confidencial do cliente e enviá-la para o servidor. Ele recebe a chave pública do servidor e usa ela para gerar a chave compartilhada, a envia novamente para o servidor, e recebe a chave compartilhada do servidor.

```
#Recebe Chave/Senha de criptografia
data3 = mClientSocket.recv(2048)
req3 = data3.decode()
senhaCriptografia = req3

#Cria assinatura e manda para o servidor
assinaturaDig = random.randint(0,999999999)
assinatura = f"{assinaturaDig}"
mClientSocket.send(assinatura.encode())
print(assinatura)
```

Depois de receber a chave compartilhada do servidor, o cliente recebe a chave de criptografia, cria uma assinatura digital por meio da biblioteca random, que gera um número aleatório para ser usado como assinatura digital e a retorna para o servidor a assinatura para que ele use também .

```
#Começa a transferencia dos dados e verifica se a chave compartilhada recebeida é a mesma enviada
while True and req2 == str(chaveCompartilhada):
    # Este loop foi criado apenas para que o cliente conseguisse enviar múltiplas solicitações
    mensagem = input('>>')
    #criptografa a mensagem recebida
    mensagemCriptografada = critogrtafiaAES(mensagem,senhaCriptografia)
    #Envia a mensagem criptografada pelo socket criado
    mClientSocket.send(mensagemCriptografada.encode())

    mClientSocket.send(assinatura.encode())
    #Recebendo as respostas do servidor
    data = mClientSocket.recv(2048)
    reply = data.decode()
    #Recebe assinatura
    data1 = mClientSocket.recv(2048)
    reply1 = data1.decode()

    #Verifica assinatura
    if str(reply1) != str(assinatura):
        break
    else:
        #descriptografa a mensagem recebida
        mensagemDescriptografada = descriptografiaAES(reply, senhaCriptografia)
        print(f'Resposta recebida:{mensagemDescriptografada}')
```

Aqui começa a parte do envio das mensagens. De começo, o while true para envio de múltiplas mensagens. Após o cliente digitar sua mensagem, ela é criptografada e enviada para o servidor junto com a assinatura digital. O servidor vai receber essa mensagem e vai retornar com uma mensagem de confirmação dizendo que recebeu a mensagem do cliente juntamente com seu ip, porta e a assinatura. A assinatura vai ser verificada e depois a mensagem será descriptografada e printada no terminal.

## PROJETO 1 SERVIDOR

Do lado do **servidor**, temos 3 funções, sendo 2 delas já explicadas anteriormente, que são as funções de criptografia, e a função HandleRequest.

```
listaClientes = []

mSocketServer = socket(AF_INET, SOCK_STREAM)

mSocketServer.bind(('127.0.0.1',1235))

mSocketServer.listen()

while True:
    clientSocket, clientAddr = mSocketServer.accept()
    Thread(target=HandleRequest, args=(clientSocket, clientAddr)).start()
```

Aqui é onde o servidor é iniciado, definindo o protocolo (TCP) e o endereço ip e porta, o servidor então vai ficar em “alerta” para receber conexões de possíveis clientes. Depois, temos o ‘while True’, que vai aceitar a conexão do cliente e vai rodar a função do HandleRequest que permite o processo de multiThreads.

```python
def HandleRequest(mClientSocket, mClientAddr):
    #Recebe no data1 o primo e gerador que foi enviado pelo cliente
    data1 = mClientSocket.recv(2048)

    req1 = data1.decode()
    chaves = req1.split(' ')

    primoComum = int(chaves[1])
    gerador = int(chaves[2])
    print(f'primocomum = {primoComum}// gerador = {gerador}')
    #envia para o cliente que esta de acordo com o primo e gerador recebido
    rep = 'Chaves OK'
    mClientSocket.send(rep.encode())

    #Cria uma chave confidencial do servidor
    chaveConfidencialServidor = random.randint(0,1000)
    #Executa o primeiro dieff hellman e envia para o servidor sua chave publica do cliente o rep1
    rep1 = ((gerador**(chaveConfidencialServidor)) % primoComum)
    mClientSocket.send(str(rep1).encode())

    #Recebe a chave publica do cliente
    data2 = mClientSocket.recv(2048)
    req2 = data2.decode()
    chavePublicaCliente = int(req2)
```

Essa parte do código é bem semelhante ao do cliente, já que a relação dos dois vai ser de um enviar, o outro receber, e vice-versa. O servidor vai receber o número aleatório gerado e o número primo, e, enviará para o cliente que as chaves estão OK. Vai criar a chave confidencial, fazer o Diffie Hellman, enviar para o cliente e vai receber a chave pública do cliente.

```python
# gera a chave compartihada do cliente e servidor
chaveCompartilhada = (chavePublicaCliente ** chaveConfidencialServidor) % primoComum
print(f'Chave compartilhada = {chaveCompartilhada}')
# Envia para o cliente a chave compartilhada

mClientSocket.send(str(chaveCompartilhada).encode())

# Recebe a chave compartilhada do cliente
data3 = mClientSocket.recv(2048)
req3 = data3.decode()

# Cria senha para a criptografia e manda para o cliente
senhaCriptografia = random.randint(100, 999)
SenhaCriptografiaEnviar = (f'{senhaCriptografia}')
mClientSocket.send(SenhaCriptografiaEnviar.encode())

# Recebe assinatura das mensagens
data4 = mClientSocket.recv(2048)
reqassinatura = data4.decode()
assinatura = descriptografiaAES(reqassinatura, senhaCriptografia)
```

Depois, gerará a chave compartilhada e enviará para o cliente, receberá a chave compartilhada do cliente e criará a senha para a criptografia, e, por fim, receberá a assinatura digital do cliente.

**Até então o código se resumia a o projeto 1, agora é mostrado a implementação do projeto 2**

## PROJETO 2 SERVIDOR

Primeiro vamos olhar o **servidor**. Primeiramente é importante mostrar o uso inicial da biblioteca 'fernet', gerando e guardando uma chave, e depois usando a função de criptografia fernet com base na chave gerada.

```
# geração de chave
key = Fernet.generate_key()
# string a chave em um arquivo
with open('filekey.key', 'wb') as filekey:
    filekey.write(key)

# usando a chave gerada
fernet = Fernet(key)
```

Vamos pular diretamente para essa parte, pois tudo antes disso é exatamente igual ao projeto 1, que é a parte do Diffie Hellman e troca de chaves. Com a assinatura descriptografada em mãos,  estando de acordo, o servidor vai receber a identificação do cliente junto com a assinatura da mensagem, e verificar se a assinatura é válida.

```
# verifica se a chave compartilhada recebeida é a mesma enviada, e recebe o indentificador do cliente
if req3 == str(chaveCompartilhada):
    # Recebe identificacao do cliente
    data = mClientSocket.recv(2048)
    print(f'Requisição recebida de {mClientAddr}')
    req = data.decode()

    # Recebe Assinatura da mensagem recebida
    data1 = mClientSocket.recv(2048)
    reqassinatura = data1.decode()

    # Verifica se a assinatura esta correta
    if str(reqassinatura) != str(assinatura):
        print('Assinatura incompativel')
```

Se a assinatura for válida, o servidor vai enviar para o cliente seu endereço IP criptografado e a porta que está usando. Assinando essa mensagem que contém o IP e porta

```
else:
    #Coma a assinatira Ok, ele envia para o cliente uma mensagem com o seu endereço e assina essa mensagem
    reqDescriptografado = descriptografiaAES(req, senhaCriptografia)
    print(f'Identificação do cliente: {reqDescriptografado}')
    rep = f'Seu endereço: {mClientAddr}'
    repCriptografado = critogrtafiaAES(rep, senhaCriptografia)
    mClientSocket.send(repCriptografado.encode())
    # envia assinatura
    mClientSocket.send(str(reqassinatura).encode())

    # Verifica se a indetificacao do cliente consta na lista de clientes autorizados
    if int(reqDescriptografado) in clientesAutorizados:
        # estando autorizado, envia para o cliente uma confirmação que esta autorizado
        mensagemAutorizacao = 'cliente Autorizado'
        menssagemCriptografada = critogrtafiaAES(mensagemAutorizacao, senhaCriptografia)
        mClientSocket.send(menssagemCriptografada.encode())
```

Ele também verifica se a identificação do cliente está na lista de clientes liberados, que é a seguinte lista.

```
clientesAutorizados = [22, 10, 45, 44, 4433, 222, 44777]
```

Se a identificação do cliente estiver contida dentro da lista de clientes autorizados, o servidor enviará uma confirmação para o cliente que ele está autorizado a acessar o servidor. Caso o cliente não seja autorizado a acessar, ele entra na condição de else deste if e print a mensagem de **erro 403**

```
else:
    #Se o cliente nao estiver autorizdo parta acessar o servidor, imprimi a mensagem de erro 403 e fecha a conexao com o servidor
    print(htmlMessage.erro403())
    mClientSocket.close()
```

Então, o servidor vai começar um laço de repetição while, que vai servir para o servidor receber vários arquivos em sequência, onde vai ser recebido um nome de arquivo criptografado.

```
#Inicia um loop, para receber os nomes dos arquivos e envia-los para o cliente
while True:
    #recebe nome do arquivo criptografado e descriptografava ele
    nomeArquivo = mClientSocket.recv(2048).decode()
    nomeArquivoDescriptografado = descriptografiaAES(nomeArquivo, senhaCriptografia)

    #Verifica se a extensao do arquivo é compativel
    extensao = nomeArquivoDescriptografado.split('.')[-1]
    arquivoBinario = False
    if extensao in tipoArquivoBinario:
        arquivoBinario = True
    #Verifica se a extensao do arquivo é compativel
    if (arquivoBinario is False) and extensao not in tipoArquivoText:
        #Se nao é compativel ele fecha a conexao do servidor e imprimi a mensagem de erro
        print(htmlMessage.BadRequest())
        mClientSocket.close()
```

Após entrar no laço de repetição, ele vai descriptografar o nome do arquivo recebido do cliente, e, assim, obterá o nome real do arquivo. Após isso, é verificado se o arquivo recebido é um arquivo binário ou arquivo de texto, a partir da extensão após o ponto, se não estiver em nenhuma extensão permitida, então vai ser disparado o **erro 400 Bad Request**.

```
#Verifica se a extensao do arquivo é compativel
extensao = nomeArquivoDescriptografado.split('.')[-1]
arquivoBinario = False
if extensao in tipoArquivoBinario:
    arquivoBinario = True
#Verifica se a extensao do arquivo é compativel
if (arquivoBinario is False) and extensao not in tipoArquivoText:
    #Se nao é compativel ele fecha a conexao do servidor e imprimi a mensagem de erro
    print(htmlMessage.BadRequest())
    mClientSocket.close()
else:
    #Se é compativel, é enviado a chave para a criptografia do fernet
    mClientSocket.send(key)
```

Se Não houver erros de sintaxe, então servidor vai enviar a chave de criptografia de arquivos para o cliente, e após isso vai tentar abrir o arquivo, criptografar o arquivo, e enviar o arquivo já criptografado para o cliente, se entrar na exceção de 'FileNotFoundError', então o servidor printa a mensagem html do **erro 404 Not Found**.

```
try:
    #É aberto o arquivo, que apos isso é criptografado pelo comando de fernet.encrypt e enviado criptografado para o cliente
    file = open(nomeArquivoDescriptografado, 'rb')
    original = file.read()
    encrypted = fernet.encrypt(original)
    mClientSocket.send(encrypted)
except FileNotFoundError:
    #Printa a mensagem de erro NotFound, no caso de nao encontrar o arquivo
    print(htmlMessage.NotFound())
```

Para controle do servidor, ele vai armazenando em uma lista a identificação dos clientes, junto com sua chave compartilhada, gerado no Diffie Hellman e o endereço IP e porta.

```
# Adicioana clientes na lista de clientes
if mClientAddr not in listaClientes:
    listaClientes.append([f'Identificação: {reqDescriptografado}', f'Chave compartilhada: {chaveCompartilhada}',f'endereço: {mClientAddr}'])
```

**PROJETO 2 CLIENTE**

Agora na parte do cliente, vamos mostrar o funcionamento da solicitação de arquivos e seu recebimento. Vamos direto para essa parte do código, já que assim como no servidor, tudo antes disso é praticamente igual ao projeto 1, com poucas diferenças, que já foram explicadas anteriormente.

Primeiro, o cliente envia para o server sua identificação, que seria um número que deve constar na lista de clientes autorizados, e após isso já recebe do servidor as mensagens "data" que contém uma resposta do servidor com seu address (IP e porta), e o "data1" que contém a assinatura dessa mensagem. Se assinatura não estiver Ok, ela entra na condição desse if mais abaixo.

```
if req2 == str(chaveCompartilhada):
    mensagem = input('Digite sua identificação >>')
    # criptografa a mensagem recebida
    mensagemCriptografada = critogrtafiaAES(mensagem, senhaCriptografia)
    # Envia a mensagem criptografada pelo socket criado
    mClientSocket.send(mensagemCriptografada.encode())

    mClientSocket.send(assinatura.encode())
    # Recebendo as respostas do servidor
    data = mClientSocket.recv(2048)
    reply = data.decode()
    # Recebe assinatura
    data1 = mClientSocket.recv(2048)
    reply1 = data1.decode()

    # Verifica assinatura
    if str(reply1) != str(assinatura):
        print('Assinatura incompativel')
```

Com a assinatura válida ele irá descriptografar a mensagem do server e printar no cliente, e após isso receber a mensagem de confirmação de cliente autorizado ou não pelo servidor (data2) , se não estiver autorizado, ele nao entra na condição do if e server termina a conexão com esse cliente

```
else:
    # descriptografa a mensagem recebida
    mensagemDescriptografada = descriptografiaAES(reply, senhaCriptografia)
    print(f'Menssagem servidor: {mensagemDescriptografada}')

    data2 = mClientSocket.recv(2048)
    autorizacao = data2.decode()
    autorizacaoDescriptografado = descriptografiaAES(autorizacao, senhaCriptografia)
    if autorizacaoDescriptografado == 'cliente Autorizado':
```

O cliente estando autorizado, ele irá entrar em um loop que permite múltiplas solicitações de arquivos. O comando de input recebe o nome que o cliente digita e envia pelo socket o nome do arquivo, e caso tenha algum erro ocorre aquelas mensagens que foram descritos no parte do servidor, porém caso esteja ok e o arquivo não tenha erro de sintaxe e exista no servidor, é recebido a chave da criptografia fernet, e aberto um arquivo que irá receber o arquivo descriptografado pelo comando decrypt do fernet e irá escrever em bytes neste arquivo do cliente o arquivo que o cliente solicitou ao servidor. E por fim fecha o arquivo e põe na pasta do cliente .

```
while True:
    #Recebe nome do arquivo e criptografa para enviar para o server
    nomeArquivo = input('Digite o nome do arquivo >>')
    nomeArquivoCriptografado = critogrtafiaAES(nomeArquivo, senhaCriptografia)
    mClientSocket.send(nomeArquivoCriptografado.encode())

    #Recebe a chave que criptografa os arquivos por meio da biblioteca fernet
    recebeChave = mClientSocket.recv(1024)
    recebeChave.decode()
    fernet = Fernet(recebeChave)

    #Abri um arquivo novo e recebe do servidor o arquivo criptografado e descriptografa ele com o comando decrypt, e escreve no novo arquivo
    with open(nomeArquivo, 'wb') as file:
        encriptado = mClientSocket.recv(1500000)
        descriptado = fernet.decrypt(encriptado)
        file.write(descriptado)
```

O seguinte pseudo código foi escrito para ajudar a entender o processo de envio de arquivos entre cliente e servidor.

```
(servidor)
Gera chave                                                  #biblioteca fernet cria uma chave aleatória
criptografia = chave                                        #faz a função criptografia com base na chave
servidor recebe ArquivoSolicitado
se ArquivoSolicitado != BadRequest então:
        servidor envia chave para o cliente
        ArquivoCriptografado = criptografia(ArquivoSolicitado)  #Criptografa a imagem que foi solicitada
        servidor envia ArquivoCriptografado                 #Envia um array de bytes, que é criptografado


(cliente)
escreva ArquivoSolicitado
cliente recebe chave
descriptografia = chave                                     #Cria a descriptografia com base na mesma chave
cliente recebe ArquivoCriptografado
ArquivoDescriptografado = descriptografia(ArquivoCriptografado) #aplica a função de descriptografia no array de bytes
cliente guarda ArquivoDescriptografado
```

**Sobre as Mensagens de erro**

O print dos erros é feito com um cabeçalho HTML dentro de um arquivo.py com os erros que precisamos.

```
def NotFound():
    now = datetime.now()
    mStamp = mktime(now.timetuple())

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

    html = ''
    html += '<html>'
    html += '<head>'
    html += '<title>Not Found - CIn/UFPE</title>'
    html += '<meta charset="UTF-8">'
    html += '</head>'
    html += '<body>'
    html += '<h1>Essa requisição não foi encontrada no servidor</h1>'
    html += '</body>'
    html += '</html>'
```

```
def BadRequest():
    now = datetime.now()
    mStamp = mktime(now.timetuple())

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

    html = ''
    html += '<html>'
    html += '<head>'
    html += '<title>Bad Request</title>'
    html += '<meta charset="UTF-8">'
    html += '</head>'
    html += '<body>'
    html += '<h1>Requisição não entendida pelo servidor, houve um erro de sintaxe</h1>'
    html += '</body>'
    html += '</html>'
```

```python
def erro403():
    now = datetime.now()
    mStamp = mktime(now.timetuple())

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

    html = ''
    html += '<html>'
    html += '<head>'
    html += '<title>Forbidden</title>'
    html += '<meta charset="UTF-8">'
    html += '</head>'
    html += '<body>'
    html += '<h1>Cliente não tem acesso aos arquivos do servidor</h1>'
    html += '</body>'
    html += '</html>'
```

Os arquivos são mostrados no terminal da maneira acima.

**2.2 Justificativas**:

Utilizamos a biblioteca **Random** para geração de números aleatórios para criação do número primo do Diffie Hellman, para a criação da chave confidencial do cliente e do servidor, para a senha para criptografia e para a assinatura digital.

Utilizamos também a biblioteca **Cryptocode** para realizar as criptografias necessárias.

Utilizamos a biblioteca **Thread** para os múltiplos clientes e a biblioteca **Socket** porque é necessário para programação com sockets em python, utilizando o protocolo TCP.

Criamos uma lista de possíveis identificadores liberados, porque achamos que seria mais fácil desenvolver dessa forma.

Utilizamos a biblioteca **Random** para a assinatura digital pois não conseguimos utilizar a **RSA**, então criamos nossa própria assinatura digital de um jeito mais simples, para não ficar sem esse critério.

Como não conseguimos utilizar a mesma biblioteca cryptocode para a criptografia das imagens, então está sendo feita usando a biblioteca **Fernet**.

**2.3 Resultados**

**PROJETO 1**

```
C:\Users\luizg\anaconda3\python.exe
Chave publica do servidor = 614
chave compartilhada = 129
701903669
>>
```

Print do cliente após ser inicializado e se conectar ao servidor.

```
primocomum = 661// gerador = 222
Chave compartilhada = 129
Esperando o próximo pacote ...
```

Print do servidor após o cliente estabelecer uma conexão.

```
>>testeD
Resposta recebida:Oi clinte ('127.0.0.1', 54826)
>>
```

Print do cliente após envio de uma mensagem/pacote

```
Requisição recebida de ('127.0.0.1', 54826)
A requisição foi:teste
Esperando o próximo pacote ...
```

Print do servidor após receber a mensagem/pacote.

**PROJETO 2**

```
servidor ouvindo em 127.0.0.1:1235
primocomum = 967// gerador = 477
Chave compartilhada = 237
```

Print após iniciar o servidor e estabelecer conexão com o cliente.

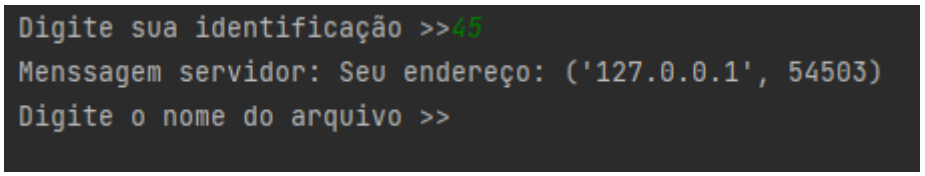
```
Chave publica do servidor = 550
chave compartilhada = 237
Digite sua identificação >>
```

Print do cliente após ser executado.

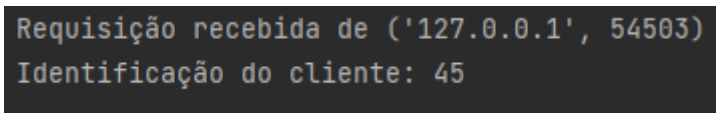
```
Digite sua identificação >>2
Menssagem servidor: Seu endereço: ('127.0.0.1', 54338)

Process finished with exit code 0
```

Print do cliente, para quando o mesmo envia uma identificação inválida e não tem acesso ao servidor.


Print do servidor após o cliente enviar uma identificação inválida.



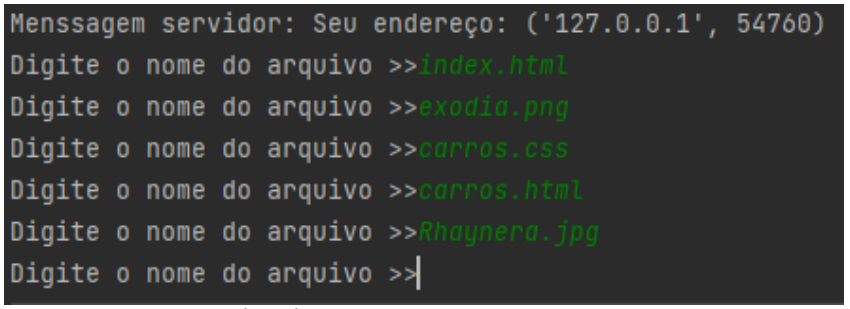

Print do cliente quando envia uma identificação válida.



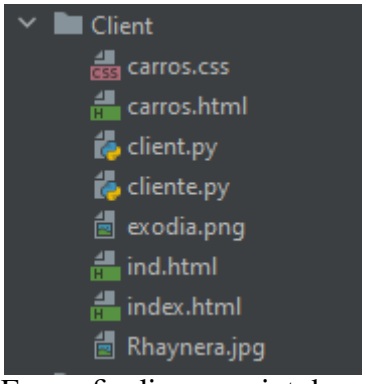

Print do servidor quando o cliente envia uma identificação válida.


Print do servidor quando o cliente solicita um arquivo inexistente(no exemplo foi digitado ind.html)


Print do servidor quando o cliente solicita uma extensão não existente(erro de sintaxe. Nesse exemplo foi digitado index.katarine)




Print do cliente após várias solicitações de arquivos. Quando o arquivo é solicitado corretamente sem ocorrer erros, o servidor não printa nada.



E para finalizar, o print da pasta Client após o cliente solicitar os arquivos corretamente.

### 2.4 Manual do usuário

Vídeo explicando como devemos rodar o código pelo Visual Studio Code upado no drive -> link

Alguns tópicos a salientar:

- Use como identificador do cliente os valores que estão válidos na lista de clientesAutorizados = [22, 10, 45, 44, 4433, 222, 44777] , caso queira ter acesso aos arquivos.

Passo a Passo:

O código se encontra no github seguinte repositório, na pasta "projeto 2" basta criar um git clone do repositório e executar os arquivos "serverer.py" e "client.py" na pasta projeto 2. Obviamente o python 3 precisa estar instalado para executar o código do cliente e servidor, é importante que o mesmo seja aberto pelo terminal, e o arquivo .py seja executado pela pasta em que o mesmo se encontra, através do comando "py *nomedoarquivo*.py", ao invés de abrir direto pela IDE e dar run.

Primeiramente, é necessário iniciar o servidor, então tem que ir no terminal, abrir a pasta do serverer.py e dar "py serverer.py" para executar o servidor.

Após iniciar o servidor, deve-se iniciar o cliente, então é só seguir os mesmos passos do servidor, porém o comando será "py client.py".

Depois de executar o cliente, a conexão com o servidor vai ser estabelecida, sendo printado na tela a chave pública do servidor, a chave compartilhada e a assinatura. O cliente então pode fazer o envio de mensagens para o servidor, e vai ser retornado no terminal a resposta do servidor, e o cliente poderá pedir quantos arquivos quiser.

## 3. Pendências

### Projeto 1

Referente aos principais critérios exigidos na elaboração do projeto, foi possível finalizar o multithread do projeto, mas não conseguimos fazer o servidor gerenciar se a

identificação que o novo cliente colocou já está sendo utilizada por outro cliente, acarretando no erro de dois clientes podendo ter a mesma identificação.

### Projeto 2

Por consequência do projeto 1 onde não conseguimos fazer gerência dos identificadores dos clientes, não é possível fazer o servidor enviar arquivos a qualquer momento para dois ou mais clientes.

Nossa principal pendência é no item H, que por conta da incompatibilidade entre a chave da criptografia fernet e a chave do cryptocode que utilizamos, acabou que tivemos que gerar uma nova chave para o fernet, que criptografa excepcionalmente os arquivos, enquanto a criptografia das mensagens trocadas entre servidor e cliente utilizaram a chave que foi feita por um número aleatório com a biblioteca random e posto como chave na criptografia do cyptocode.

Tivemos um pequeno problema na aplicação do erro 404 Not found, em que o servidor não sinaliza ao cliente esse erro, pedindo um novo arquivo.

Há outros bugs menores que não achamos resolução, como na inconsistência de algumas bibliotecas, e a criação de um arquivo vazio quando ocorre o erro 404.

## 4. Dificuldades

O desenvolvimento com sockets mostrou-se bastante complicado visto que as mensagens de erros não eram muito claras, fazendo os membros da equipe perderem muito tempo resolvendo erros(que muitas vezes acarretam outros erros), o alinhamento de “send” e “recv” ocupou a boa parte do tempo.

O armazenamento das chaves para usos futuros também não foi realizado, não encontramos uma solução para isso.

Ao aplicar as assinaturas digitais para todas as mensagens mais ao final acabou prejudicando o desempenho do servidor, aparecendo erros em que o cliente não respondia mais ao servidor. Então, preferimos por manter o código sem ter todas as assinaturas, para manter o funcionamento correto do servidor de forma constante, o arquivo sem todas as assinaturas esta no github como “serverer.py ” e “client.py”, ja o com as assinaturas todos, porém de forma inconsistente esta no github como “servidor.py” e “cliente.py”.

Por último, o código apresentou falhas em relação à biblioteca "fernet", utilizada com o propósito de codificar os dados enviados pelo servidor para o cliente.

Infelizmente, devido a complexidade do assunto e limitações das informações pesquisadas nas redes, não conseguimos resolver essa vertente com 100% de aproveitamento.

Ao aplicar as assinaturas digitais para todas as mensagens mais ao final acabou prejudicando

## 5. Conclusão

Foi possível, a partir do desenvolvimento de ambas as partes do projeto como um todo, entender como funciona de fato a relação entre Servidor-Cliente quando existe alguma identificação, que nesse caso é o Diffie-Hellman. Apesar de não conseguirmos realizar 100% das ideias pedidas do projeto, essa foi a primeira experiência de todos os membros da equipe com a criptografia e a programação utilizando sockets. Dado esse ponto, sentimos a evolução de todos os membros durante a elaboração do trabalho.

O trabalho, como um todo, levou todos os membros a buscarem cada vez mais conhecimento e adicionou um âmbito novo de tecnologia da informação para desenvolvermos melhor.

Além disso, foi possível entendermos as relações do projeto com o âmbito e o funcionamento atual das redes. Seria muito simples elaborarmos os processos de servidor-cliente e cliente-servidor utilizando módulos já estabelecidos de bibliotecas do python 3 (http-Server, por exemplo). Mas a ideia era entendermos a forma manual de como um servidor funciona e age com seus clientes.