

Comunicações por Computador

TP2 - Rede Overlay de Anonimização do Originador

[PL1] - Grupo 1

João Coutinho
(a86272)

Moisés Antunes
(A82263)

Rafael Lourenço
(A86266)

Índice

1	Introdução	3
2	Arquitectura da solução	4
2.1	Tratamento de pedidos de clientes	4
2.2	Tratamento de pedidos de servidores <i>AnonGW</i>	5
2.3	Recepção de dados do servidor	6
2.4	Envio de pacotes aos clientes	7
3	Especificação do protocolo UDP	9
4	Implementação	10
4.1	Parâmetros, detalhes, e bibliotecas	10
4.2	Encriptação	12
5	Testes e Resultados	13
6	Conclusões e trabalho futuro	17

1 Introdução

No âmbito da unidade curricular de Comunicações por Computador, foi proposto desenhar um sistema de comunicação entre um cliente e um servidor cujo objectivo é a anonimização do originador do pedido ao servidor. Este relatório visa detalhar uma implementação em *Python* de uma possível solução para este problema. Nesta solução, a rede de anonimização é composta por várias instâncias de um servidor anonimizador, doravante *AnonGW*, que, através da comunicação entre essas várias instâncias, camuflam a origem do pedido e entregam os dados de forma cifrada.

2 Arquitectura da solução

A rede anonimizadora é composta por várias instâncias de *AnonGW*. Quando um cliente faz um pedido a um dos nós da rede, esse nó, guardando localmente em memória a origem do pedido, envia os detalhes a um outro nó da rede, ocultando as informações que expõe o cliente. Esse segundo nó faz o pedido ao servidor e recebe de volta a informação pretendida, reencaminhando, pacote a pacote, imediatamente de volta ao nó anterior, de forma ordenada, garantindo que os pacotes cheguem ao seu destino correctamente.

A comunicação entre os vários agentes que interagem nesta rede é representada da seguinte forma:

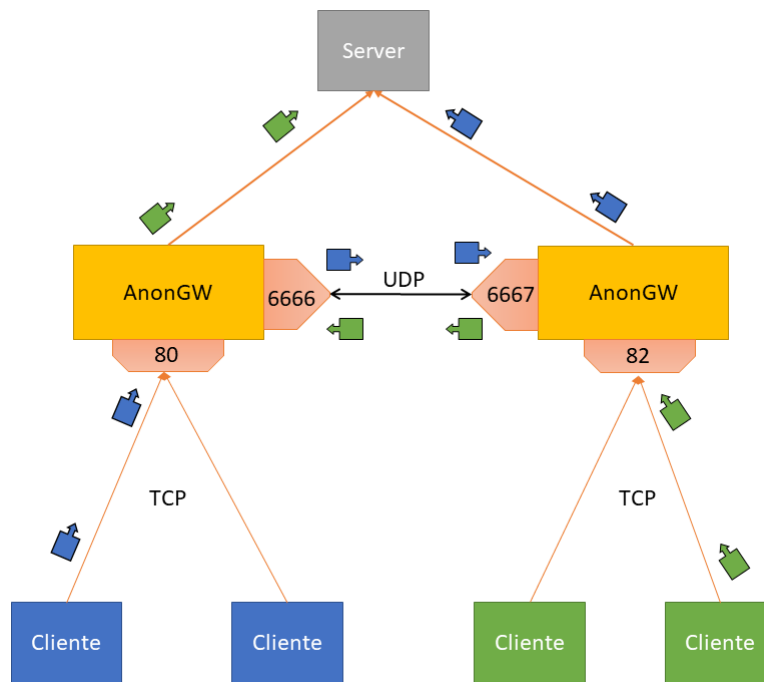


Figura 1: Diagrama de comunicação entre clientes, nós da rede, e servidor alvo

2.1 Tratamento de pedidos de clientes

Quando um nó da rede é inicializado, este lança imediatamente uma *thread* que cria uma *socket* TCP. Após a sua criação, fica à espera de receber ligações vindas de um cliente. No momento em que é feita uma ligação, é lançada uma *thread* para lidar com o pedido a caminho.

```

def initTcpSocket():
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        h = parsePeer(HOST)
        s.bind((h[0], h[1]))
        s.listen(40)
        print('\tAnonGW Disponivel!')
        while True:
            conn, addr = s.accept()
            x = threading.Thread(target=receberPedidoCli,
                                args=(conn, addr, next_id()))
            x.start()

```

Figura 2: Criação da *socket* TCP e lançamento da *thread* que vai lidar com o pedido de um cliente

A *thread* responsável pelo tratamento de um pedido de um cliente recebe, em primeiro lugar, o pedido em si, e aproveita esse momento para guardar num dicionário seu uma entrada cuja chave é um identificador desse cliente e cujo valor correspondente é um triplo composto pela *socket* que o liga ao cliente, o endereço respectivo, e um acumulador que será utilizado num trabalho futuro para efeitos de retransmissão de pacotes perdidos.

Este dicionário permite que o nó que recebeu o pedido consiga manter um registo seu, nunca partilhado com outros nós da rede, e guardado unicamente em memória. Este registo será usado posteriormente para enviar de volta os dados vindos do nó a que se conectou referentes ao pedido desse cliente.

```

def receberPedidoCli(conn, addr, id_cli):
    data = conn.recv(4096)
    clientId[id_cli] = (conn, addr, 0)
    anon = peer[randint(0, (len(peer) - 1))]
    f = Fernet(key_peer[anon[1]])
    data_cipher = f.encrypt(data)
    sig = crypt.signing(str(PORT_UDP), data_cipher)
    pacote = tgl.Header(sig, 1, 0, id_cli, 1, PORT_UDP, data_cipher)
    pacBin = pacote.converte()
    resp = enviarPedidoAGW(pacBin, anon)

```

Figura 3: Tratamento do pedido de um cliente e respectivo envio a um *AnonGW*

De seguida, é criado um pacote de pedido (*flag isQuery* a 1) e esse pedido é enviado pela *socket* UDP a um nó aleatório da rede. Enviado este pacote, esta *thread* fica a ler dados vindos dessa *socket* até obter resposta.

2.2 Tratamento de pedidos de servidores *AnonGW*

Como mencionado anteriormente, subsequentemente a ser lançada uma *thread* para lidar com pedidos vindos de clientes, é criada uma *socket* UDP. Após um processo, discutido mais à frente, em que a rede trata da partilha de informação necessária à encriptação e desencriptação da informação transmitida, o nó fica à escuta nesta *socket* por pedidos vindos de outros nós da rede. Chegando um pedido, é lançada

uma *thread* que o tratará, enquanto que o processo pai fica à escuta de mais pedidos, repetindo o processo indefinidamente. É através deste método que é garantido o processamento de pedidos de vários clientes simultaneamente.

```
UDPServerSocket = socket.socket(family = socket.AF_INET, type = socket.SOCK_DGRAM)
UDPServerSocket.bind((h[0], PORT_UDP))
time.sleep(2)
trocarChaves(MY_key)
i=1
while True:
    (data,addr) = UDPServerSocket.recvfrom(4096)
    i+=1
    pacote = tgl.desconverte(data)
    if(pacote.getNumPed()==0):
        crypt.verificaton(str(pacote.getPort()), pacote.getSignature(), pacote.getMsg())
        key = crypt.decrypt(pacote.getMsg(), str(PORT_UDP))
        key_peer[pacote.getPort()] = key
    else:
        y=threading.Thread(receberPedidoAnon, (UDPServerSocket, addr, pacote))
        y.start()
```

Figura 4: Criação da *socket* UDP e tratamento do pedido recebido

Caso o campo **n_ped** seja igual a 0, então o pedido recebido é um pedido de troca de informação criptográfica, ao que o nó procede a verificar a assinatura do pacote, à descriptação da informação contida, e à adição do nó correspondente ao seu dicionário de nós. Caso contrário, cria e lança uma *thread* para lidar com o pedido recebido.

```
def receberPedidoAnon(UDPServerSocket,addr,pacote):
    try:
        crypt.verificaton(str(pacote.getPort()),pacote.getSignature(),pacote.getMsg())
        plain_data=MY_fernet.decrypt(pacote.getMsg())
        pacote_plain = tgl.Header(pacote.getSignature(), pacote.get_isQuery(),
        pacote.is_ultimoPac(), pacote.getCliente(), pacote.getNumPed(), pacote.getPort(),
        plain_data)
        res = enviarServ(ServPORT,pacote_plain,UDPServerSocket,addr
```

Figura 5: Tratamento do pedido recebido de um nó

O pedido recebido é de seguida enviado ao servidor alvo através de uma *socket* TCP. Tendo sido enviada pelo servidor a informação pretendida, o nó lê a informação recebida, um pacote de cada vez. Lendo um pacote, envia-o ao nó que lhe fez o pedido, repetindo o processo até ser concluído o envio da informação toda. Terminado o envio do último pacote, a *thread* termina a execução, fechando o *socket* TCP.

2.3 Recepção de dados do servidor

Enviada a informação ao servidor, o nó fica em leitura à espera que o servidor responda.

```

def enviarServ(port,pacote,UDPServerSocket,addr):
    n_ped=1
    h=parsePeer(HOST)
    sig="asdfhasdkjfaskjndafskljnfalasdda"
    try:
        ss = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        ss.connect((h[0], port))
        ss.sendall(pacote.getMsg())
        destino=pacote.getPort()
        while True:
            dados = ss.recv(Tam_PACK)
            if not dados:
                pacote2 = tgl.Header(sig, 0, 1, pacote.getClient(),
                                     n_ped, PORT_UDP, dados)
                pacBin2 = pacote2.converte()
                UDPServerSocket.sendto(pacBin2,addr)
                break
            f = Fernet(key_peer[destino])
            data_cipher = f.encrypt(dados)
            sig = crypt.signing(str(PORT_UDP),data_cipher)
            pacote = tgl.Header(sig, 0, 0, pacote.getClient(), n_ped,
                                PORT_UDP, data_cipher)
            pacBin = pacote.converte()
            n_ped += 1
            UDPServerSocket.sendto(pacBin,addr)

```

Figura 6: Envio, recepção, e tratamento da informação vinda do servidor

Recebendo um pacote, verifica primeiro se esse pacote é o último a ser recebido, enviando o pacote e terminando a conexão. Este pacote actua como um pacote "nulo", sinalizando o fim dos dados. Havendo dados para continuar a receber e enviar, o nó encripta-os, assina-os, converte-os para um pacote, e envia o pacote ao nó correspondente. Continua, enviando toda a informação pacote a pacote, pela ordem em que é recebida, até completar o pedido.

2.4 Envio de pacotes aos clientes

Como mencionado anteriormente, após um nó enviar um pedido a um outro nó, fica em leitura da *socket* UDP à espera de receber os dados correspondentes.

```

def enviarPedidoAGW(msg,peer_addr):
    try:
        sp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sp.sendto(msg, (peer_addr[0], peer_addr[1]))
        while True:
            (dados,adr) = sp.recvfrom(Tam_PACK * 8 + Tam_Header)
            pacote = tgl.desconverte(dados)
            (conn, addr, acc) = clientId[pacote.getCliente()]
            if(pacote.is_ultimoPac() == 1):
                conn.close()
                break
            crypt.verification(str(pacote.getPort()), pacote.getSignature(),
            pacote.getMsg())
            msg_plain = MY_fernet.decrypt(pacote.getMsg())
            conn.sendto(msg_plain, addr)

```

Figura 7: Envio do pedido de um cliente a um nó da rede e respectiva recepção do conteúdo correspondente

Da mesma forma que o envio da informação vinda do servidor por um nó a um outro nó é feita pacote a pacote, também a sua leitura aqui é feita pacote a pacote. Lendo um pacote da *socket* UDP, o nó desconverte-o e vai buscar ao seu dicionário a informação referente ao cliente ao qual tem de enviar os dados. Caso o pacote em questão seja o último pacote, a conexão é imediatamente encerrada e o envio fica concluído (de notar que este último pacote é o pacote "nulo"referenciado acima). Caso contrário, verifica a assinatura do nó que lhe enviou o pacote, descripta-o, e envia-o ao cliente respectivo.

3 Especificação do protocolo UDP

As mensagens protocolares são especificadas numa classe *Header*.

Protocolo AnonGw

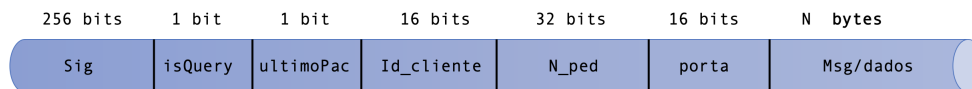


Figura 8: Estrutura de cada PDU

O campo **Sig** é a assinatura digital do AnonGW origem, que tem como objetivo garantir a autenticação do originador. Deste modo, só são aceites pacotes de origem conhecida.

O campo **isQuery** é usado como uma *flag* que determina se o pacote enviado é um pedido ao servidor ou um pacote de dados de resposta a um pedido já feito. O campo *ultimoPac* é também uma *flag* que nos indica se o pacote em questão é o último pacote referente aos dados do pedido.

O campo **ultimoPac** é usado para distinguir o último pacote dos restantes, para fechar a conexão.

O campo **id_cliente** indica qual o cliente a que o pacote diz respeito. De notar que este campo age apenas como um identificador interno do programa, não pondo em causa a identidade "real" do cliente, pois apenas o nó que recebeu o pedido de um cliente tem consigo o dicionário que contém a conexão, essa, sim, a "verdadeira" identidade, com o cliente.

O campo **n_ped** indica o número identificador de um dado pacote, e, por fim, o campo *msg* representa o conteúdo propriamente dito pedido pelo cliente.

O campo **porta** serve para identificar o AnonGW que enviou o pacote. Dado que é permitido multicast e o socket UDP é partilhado por várias threads, foi necessário saber qual era o AnonGW destino para cada pacote, portanto foi colocado no cabeçalho de cada pacote a porta UDP do AnonGW que recebeu o pedido, resolvendo assim esse problema. Por outro lado, é utilizado na parte da criptografia, uma vez que, todos os pacotes são verificados com a chave pública do originador, armazenada num ficheiro que é identificado pela sua porta **UDP** e criptografados com uma chave simétrica, guardada numa hash, explicada na próxima secção.

Por último, temos o campo **Msg/dados**, que é onde situada parte da resposta cifrada, impedindo algum intruso de a espiar. Esta tem como tamanho default 1024 bytes, porém pode ser facilmente alterada, uma vez que, esta informação é armazenada numa variável global *Tam_PACK*

4 Implementação

4.1 Parâmetros, detalhes, e bibliotecas

Aquando da execução de um servidor *AnonGW*, é passado como argumento um endereço, um porto, e um conjunto de endereços. O primeiro endereço diz respeito a uma *socket* TCP para a qual um cliente direcciona o seu pedido. O porto, em conjunto com o endereço retirado do argumnto anterior, representa a *socket* UDP onde este servidor fica à escuta de pedidos vindos de outros nós. O conjunto de endereços recebidos como restantes argumentos são os endereços de escuta dos outros nós da rede, por onde esses nós receberão os pedidos.

```
py AnonGW.py 127.0.0.1:80 6666 127.0.0.1:6667  
py AnonGW.py 127.0.0.1:82 6667 127.0.0.1:6666
```

Figura 9: Exemplo de execução de dois servidores

```
HOST = sys.argv[1]  
PORT_UDP=int(sys.argv[2])  
peer=[]  
clientId=dict()  
key_peer=dict()  
ServPORT=8000  
Tam_PACK=1024  
Tam_Header=322  
Clientes=1  
iid_lock = threading.Lock()  
pgid = os.getppid()
```

Figura 10: Tratamento dos argumentos, inicialização de várias estruturas, e declaração de valores de referência no início da execução de um servidor

```
MY_key = Fernet.generate_key()  
MY_fernet = Fernet(MY_key)
```

Figura 11: Criação da chave simétricas

```
for x in range(3, len(sys.argv)):  
    aux =parsePeer(sys.argv[x])  
    if (aux!=("-1",-1)):  
        peer.append(aux)
```

Figura 12: Armazenamento da informação dos servidores da rede

Foi ainda definido um sinal e a respectiva função de tratamento.

```
def signal_handler(sig, frame):
    print('\n\tAnonGW Fechado!')
    os.killpg(pgid, signal.SIGKILL)
```

Figura 13: Definição do sinal para término de um *AnonGW*

Foram utilizadas as seguintes **bibliotecas**:

- **socket** - Utilizada para criação e utilização de *sockets*
- **threading** - Utilizada para criar e lançar *threads* de execução do programa
- **random** - Utilizada para fazer a selecção do *AnonGW* ao qual um outro *AnonGW* envia um pedido que um cliente lhe fez
- **cryptography** - Usada para cifrar, decifrar, verificar, assinar e lancar exceções relativamente à transmissão dos pacotes.
- **signal** - Usada para manipulação de sinais(Especificamente o CRL-C)
- **os** - Usada para obtenção do *PID* do processo pai para terminar todas as threads.

4.2 Encriptação

Neste modulo, usamos criptografia assimétrica e simétrica. Dado que a confidencialidade é uma das propriedades mais importantes nos dias de hoje, baseámo-nos num método/protocolo chamado Station-to-Station(STS), que consiste em usar uma chave simétrica, acordada entre duas entidades que querem estabelecer uma comunicação. Esta é usada para cifrar e decifrar os pacotes de uma conexão nos dois sentidos. A chave simétrica é previamente cifrada e enviada por criptografia de chave publica, garantindo confidencialidade.

De forma a asseverar ainda mais segurança, decidimos que cada *AnonGW* conhecido deveria ter uma chave simétrica, onde estas são trocadas antes de indicar a recepção de clientes. Assim, quando enviávamos um pacote para um *AnonGW* destino teríamos de encriptar com a sua chave simétrica, e quando este nos queria responder teria de encriptar com a nossa chave simétrica.

Estas serão transportadas num socket UDP, com o protocolo que descrevemos acima, e cifrado com criptografia de chave pública. Este pacote onde está contida a chave simétrica também é assinado pela entidade conhecida.

De seguida, mostramos um exemplo entre 2 *AnonGW*, de combinação de chave:

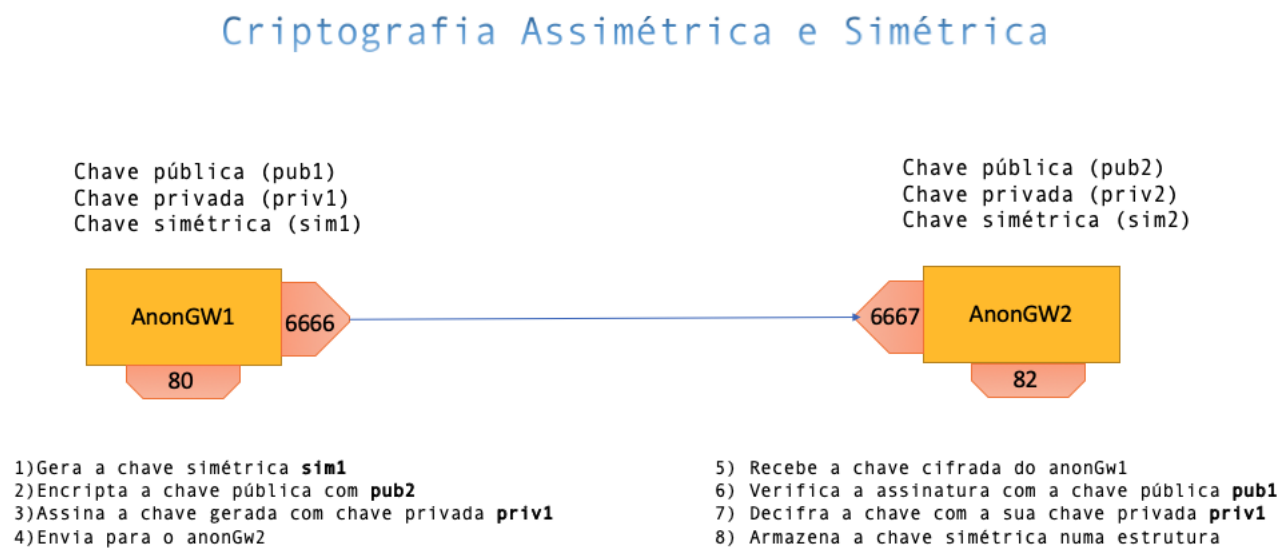
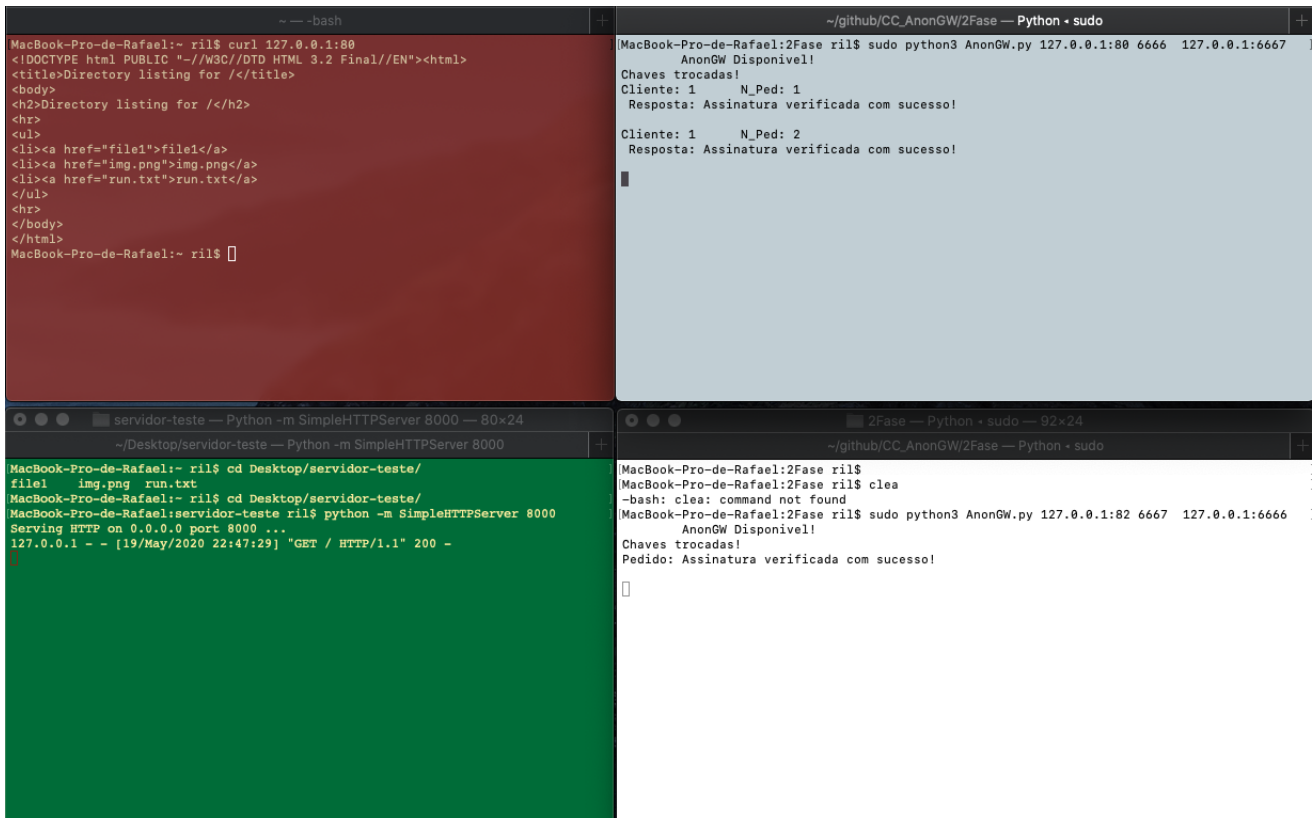


Figura 14: Troca de chaves simétricas

5 Testes e Resultados

Em primeiro lugar, podemos verificar que as chaves foram trocadas. De seguida podemos ver que o *AnonGW 6667* recebeu com sucesso o pedido ao *AnonGW 6666*, e por último, que o *AnonGW 6667* enviou a resposta para o *6666*, onde se verifica que foram entregues com sucesso.



```
MacBook-Pro-de-Rafael:~ ril$ curl 127.0.0.1:80
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>
<title>Directory listing for /</title>
<body>
<h2>Directory listing for /</h2>
<hr>
<ul>
<li><a href="file1">file1</a>
<li><a href="img.png">img.png</a>
<li><a href="run.txt">run.txt</a>
</ul>
<hr>
</body>
</html>
MacBook-Pro-de-Rafael:~ ril$

~/github/CC_AnonGW/2Fase — Python • sudo
MacBook-Pro-de-Rafael:2Fase ril$ sudo python3 AnonGW.py 127.0.0.1:80 6666 127.0.0.1:6667
AnonGW Disponivel!
Chaves trocadas!
Cliente: 1      N_Ped: 1
Resposta: Assinatura verificada com sucesso!

Cliente: 1      N_Ped: 2
Resposta: Assinatura verificada com sucesso!

servidor-teste — Python -m SimpleHTTPServer 8000 — 80x24
~/Desktop/servidor-teste — Python -m SimpleHTTPServer 8000
MacBook-Pro-de-Rafael:~ ril$ cd Desktop/servidor-teste/
file1  img.png  run.txt
MacBook-Pro-de-Rafael:~ ril$ cd Desktop/servidor-teste/
MacBook-Pro-de-Rafael:servidor-teste ril$ python -m SimpleHTTPServer 8000
Serving HTTP on 0.0.0.0 port 8000 ...
127.0.0.1 - - [19/May/2020 22:47:29] "GET / HTTP/1.1" 200 -

2Fase — Python • sudo — 92x24
~/github/CC_AnonGW/2Fase — Python • sudo
MacBook-Pro-de-Rafael:2Fase ril$
MacBook-Pro-de-Rafael:2Fase ril$ clea
-bash: clea: command not found
MacBook-Pro-de-Rafael:2Fase ril$ sudo python3 AnonGW.py 127.0.0.1:82 6667 127.0.0.1:6666
AnonGW Disponivel!
Chaves trocadas!
Pedido: Assinatura verificada com sucesso!
```

Figura 15: Teste com dois *AnonGW* recorrendo ao comando *curl*

Para um segundo conjunto de testes mais exigentes, foi desenhado um programa que executa, concorrentemente, dez *threads* a executar um pedido a um nó e outras dez a outro nó.

```

def cli():
    BODY = ""
    conn = http.client.HTTPConnection("127.0.0.1", 80)
    a=conn.request("GET", "/file1", BODY)
    response = conn.getresponse()
    aux=response.read(2000)

def cli2():
    BODY = "***filecontents***"
    conn = http.client.HTTPConnection("127.0.0.1", 81)
    conn.request("GET", "/", BODY)
    response = conn.getresponse()

```

Figura 16: Clientes de teste

```

for x in range(0,10):
    y=threading.Thread(target=cli, args=())
    y2=threading.Thread(target=cli2, args=())
    y.start()
    y2.start()

```

Figura 17: Execução concorrente dos clientes

```
~/github/CC_AnonGW/2Fase -- -bash
R2z5RTTHRJH4JXqqQtWIUGsv8Nwt1SnaBm5CH7Kiq-zsuaZYrRWJ2Dz6E0'

Cliente2 b'<DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>\n<title>Directory listing for /</title>\n<body>\n<h2>Directory listing for /</h2>\n<hr>\n<ul>\n<li><a href="file1">file1</a>\n<li><a href="img.png">img.png</a>\n<li><a href="run.txt">run.txt</a>\n</ul>\n<hr>\n</body>\n</html>\n'

Cliente1: b'https://www.youtube.com/watch?v=to8UbcNVVYc&feature=share&fbclid=IwAR2z5RTTHRJH4JXqqQtWIUGsv8Nwt1SnaBm5CH7Kiq-zsuaZYrRWJ2Dz6E0'

Cliente1: b'https://www.youtube.com/watch?v=to8UbcNVVYc&feature=share&fbclid=IwAR2z5RTTHRJH4JXqqQtWIUGsv8Nwt1SnaBm5CH7Kiq-zsuaZYrRWJ2Dz6E0'

Cliente2 b'<DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>\n<title>Directory listing for /</title>\n<body>\n<h2>Directory listing for /</h2>\n<hr>\n<ul>\n<li><a href="file1">file1</a>\n<li><a href="img.png">img.png</a>\n<li><a href="run.txt">run.txt</a>\n</ul>\n<hr>\n</body>\n</html>\n'

Cliente1: b'https://www.youtube.com/watch?v=to8UbcNVVYc&feature=share&fbclid=IwAR2z5RTTHRJH4JXqqQtWIUGsv8Nwt1SnaBm5CH7Kiq-zsuaZYrRWJ2Dz6E0'

Cliente2 b'<DOCTYPE html PUBLIC "-//W3C//DTD HTML 3.2 Final//EN"><html>\n<title>Directory listing for /</title>\n<body>\n<h2>Directory listing for /</h2>\n<hr>\n<ul>\n<li><a href="file1">file1</a>\n<li><a href="img.png">img.png</a>\n<li><a href="run.txt">run.txt</a>\n</ul>\n<hr>\n</body>\n</html>\n'

Pedido: Assinatura verificada com sucesso!

Cliente: 11      N_Ped: 1
Resposta: Assinatura verificada com sucesso!

Cliente: 13      N_Ped: 1
Resposta: Assinatura verificada com sucesso!

Cliente: 12      N_Ped: 1
Resposta: Assinatura verificada com sucesso!

Cliente: 11      N_Ped: 2
Resposta: Assinatura verificada com sucesso!

Cliente: 15      N_Ped: 1
Resposta: Assinatura verificada com sucesso!

Cliente: 14      N_Ped: 1
Resposta: Assinatura verificada com sucesso!

Cliente: 13      N_Ped: 2
Resposta: Assinatura verificada com sucesso!

Cliente: 14      N_Ped: 2

servidor-teste -- Python -m SimpleHTTPServer 8000 -- 80x24
~/Desktop/servidor-teste -- Python -m SimpleHTTPServer 8000

127.0.0.1 -- [19/May/2020 22:58:55] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:58:55] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:58:55] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:24] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET /file1 HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET / HTTP/1.1" 200 -
127.0.0.1 -- [19/May/2020 22:59:33] "GET / HTTP/1.1" 200 -

2Fase -- Python - sudo -- 174x10
~/github/CC_AnonGW/2Fase -- Python - sudo

[MacBook-Pro-de-Rafael:2Fase ril$ sudo python3 AnonGW.py 127.0.0.1:82 6668 127.0.0.1:6667 127.0.0.1:6666
AnonGW Disponivel!
Chaves trocadas!
Pedido: Assinatura verificada com sucesso!

Pedido: Assinatura verificada com sucesso!

Pedido: Assinatura verificada com sucesso!
```

Figura 18: Teste com três *AnonGW* recorrendo ao programa acima

Por último, temos um teste exaustivo, uma vez que o pedido mostrado abaixo levou à criação de 109 pacotes encapsulados no protocolo *AnonGW*. Por outro lado, mostramos os ficheiros existentes no servidor, para se poder confirmar a veracidade do pedido.

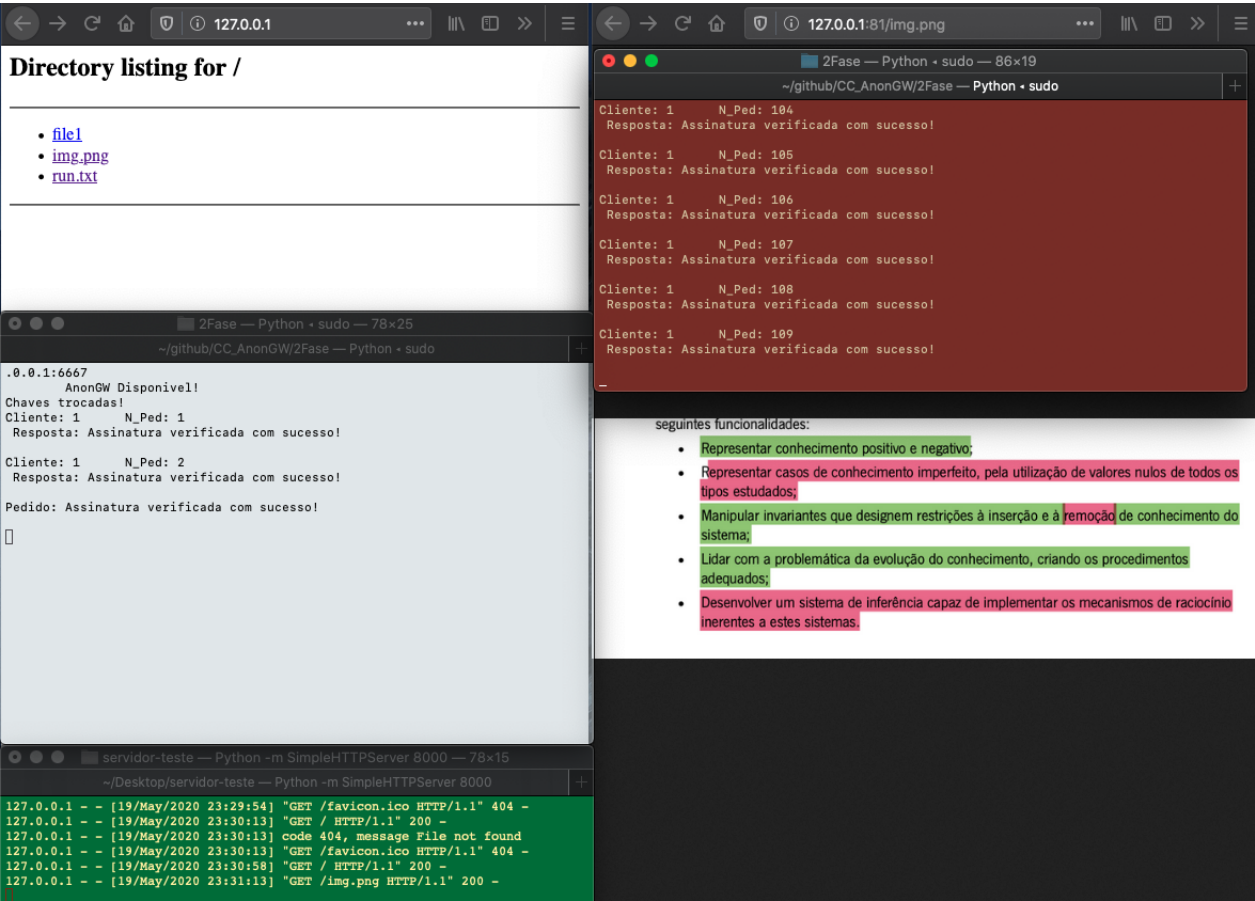


Figura 19: Testes exaustivo ao *AnonGW*, pedindo uma imagem.

6 Conclusões e trabalho futuro

No final deste exercício, a equipa conclui que foi desenvolvido com sucesso um sistema de comunicação entre um cliente e um servidor que assegura a anonimidade do cliente. Foi implementada a entrega ordenada de dados, a cifragem do conteúdo, e também a multiplexagem de clientes. Foi ainda implementado um mecanismo de autenticação da origem dos dados. Com este trabalho foi possível consolidar os conhecimentos leccionados ao longo da unidade curricular e aplicá-los numa área de especial relevo nos dias correntes.

Como trabalho futuro, a equipa destaca a implementação de um mecanismo de retransmissão de pacotes perdidos. Poderia ainda ser implementado um algoritmo de troca de informação criptográfica mais eficiente que não tivesse de recorrer à utilização de *sleeps*. De resto, considera-se a solução aqui apresentada como muito satisfatória face aos requisitos pedidos.