

Trabalho de Aprofundamento 2

Universidade de Aveiro

Mariana Silva, Marta Oliveira



VERSAO 1

Trabalho de Aprofundamento 2

Departamento de Eletrónica, Telecomunicações e
Informática

Universidade de Aveiro

Mariana Silva, Marta Oliveira
(98392) marianabarbara@ua.pt, (97613) marta.alex@ua.pt

09 de maio de 2020

Contents

1	Introdução	2
2	Código	3
2.1	Terminal	3
2.2	Explicação do código b64.py	5
2.3	Explicação do código cifradecifra.py	6
2.4	Explicação do código tcp	7
2.5	Explicação do Algoritmo do server_tcp	8
2.6	Explicação do Algoritmo do client_tcp	11
3	Análise dos Resultados Obtidos	14
4	Contribuição dos Autores	15

List of Figures

1	Terminal 1	3
2	Terminal 2	3
3	Terminal 3	4
4	Terminal 1 após comando	4
5	Terminal 2 após comando	4
6	Código base64	5
7	Código cifra/decifra	6
8	Código tcp	7
9	Função main do server	8
10	Sockets	8
11	Função new_process	9
12	Função new_client	9
13	Função list_clients	9
14	Função run_sorting	10
15	Função main	11
16	Função Client	12
17	Continuação da Função Client	12
18	Função dump_csv	13

1 Introdução

O tema proposto deste trabalho consiste em seriar, de forma aleatória, n clientes. De forma a que este serviço seja gerado de forma isenta, as aplicações cliente irão usar interfaces abertas.

Resumidamente:

Iremos ter uma interface/aplicação cliente-servidor.

Neste código, iremos utilizar os fundamentos da Interface de Programação de Aplicações (API) de Sockets.

Aplicação Cliente:

1. Criar o socket (função socket);
2. A seguir, criar uma função connect para conectar com o servidor;
3. Enviar/Receber dados enquanto existem dados para enviar/receber;
4. Fechar o socket.

Aplicação Servidor:

1. Criar o socket (função socket);
2. Inserir um endereço Protocolo da Internet (IP) e uma porta no socket (função bind);
3. A função accept serve para aceitar uma nova conexão;
4. Fechar o socket.
5. Podem ser usadas várias conexões, por isso o passo 3 pode ser repetido;

Os passos do servidor são feitos usando o Protocolo de Controlo de Transmissão (TCP)

Para além disso, neste relatório, iremos explicar a implementação não só do código como também do algoritmo, apresentar testes que comprovam o seu funcionamento e analisar os resultados obtidos.

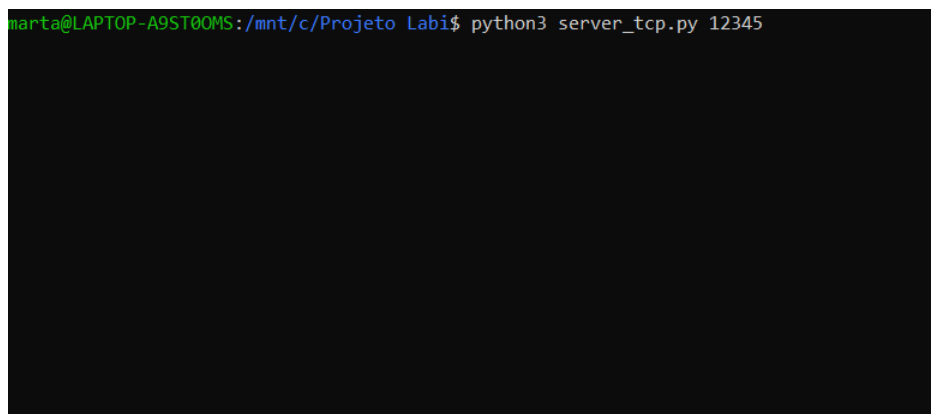
Este Trabalho de Aprofundamento foi criado no code.ua.pt com o nome labi2020-ap2-g45 (<https://code.ua.pt/projects/labi2020-ap2-g45>).

2 Código

Começámos por executar a simulação da aplicação cliente servidor para confirmar se tudo funcionava como previsto (testes iniciais).

2.1 Terminal

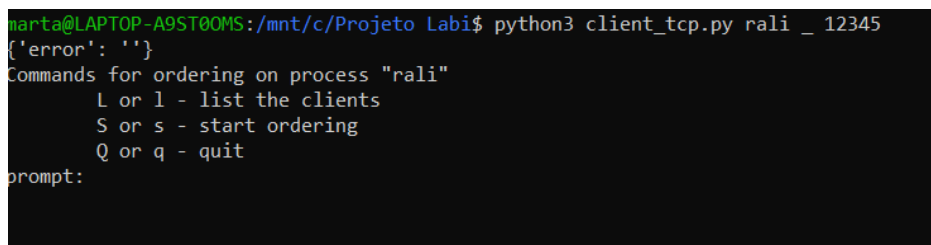
Quando ficheiro do server é iniciado, o terminal não mostra nenhuma mensagem como seria esperado.



```
marta@LAPTOP-A9ST00MS:/mnt/c/Projeto Labi$ python3 server_tcp.py 12345
```

Figure 1: Terminal 1

Ao iniciarmos o segundo terminal e ao executar o ficheiro do cliente iremos encontrar o seguinte “menu” no terminal.



```
marta@LAPTOP-A9ST00MS:/mnt/c/Projeto Labi$ python3 client_tcp.py rali _ 12345
{'error': ''}
Commands for ordering on process "rali"
  L or l - list the clients
  S or s - start ordering
  Q or q - quit
prompt:
```

Figure 2: Terminal 2

Como não temos participantes temos que iniciar um terceiro terminal e colocar 10 participantes (usando a Shell script- run_tcp).

```
marta@LAPTOP-A9ST00MS:/mnt/c/Projeto Labi$ bash run_tcp.sh rali 12345 10
```

Figure 3: Terminal 3

Logo após lançar este comando, o terminal 1 aparece da seguinte forma (uma mensagem de pedido a cada cliente)

```
marta@LAPTOP-A9ST00MS:~$ cd /mnt/c
marta@LAPTOP-A9ST00MS:/mnt/c$ cd Projeto\ Labi/
marta@LAPTOP-A9ST00MS:/mnt/c/Projeto Labi$ python3 server_tcp.py
ERROR: Invalid parameters <server port> <optional server ip>
marta@LAPTOP-A9ST00MS:/mnt/c/Projeto Labi$ python3 server_tcp.py 12345
Command: {'op': 'NEW', 'proc': 'rali'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u10'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u9'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u8'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u7'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u6'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u5'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u4'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u3'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u2'}
Command: {'op': 'ADD', 'proc': 'rali', 'id': 'u1'}
```

Figure 4: Terminal 1 após comando

Para além disso, também nos apercebemos que se os dados forem mal inseridos no terminal irá aparecer uma mensagem de erro onde pede a forma correta de inserir as “informações”.

Ao voltarmos ao terminal 2 e ao carregarmos no L (para listarmos clientes), o programa já nos irá lançar os 10 clientes.

```
prompt: l
10 clients:
    u10
    u9
    u8
    u7
    u6
    u5
    u4
    u3
    u2
    u1
```

Figure 5: Terminal 2 após comando

2.2 Explicação do código b64.py

```
import base64
from Crypto.Hash import SHA256

msg = 'This is a message that is going to be hashed with SHS-256'

hash_f = SHA256.new()
hash_f.update( bytes(msg, 'utf8') )
digest = hash_f.digest()

b64_digest = base64.b64encode( digest )
recovered_digest = base64.b64decode( b64_digest )

if digest == recovered_digest:
    print( 'Success!' )
else:
    print( 'Failure, %s is different from %s' % (digest, recovered_digest) )
```

Figure 6: Código base64

Neste código, o SHA-2 irá funcionar com funções hash.

A função hash basicamente, resume dados. Vai buscar elementos/dados de qualquer tamanho e transforma-os em dados de comprimentos fixos.

Sendo assim, o resultado da função hash irá ser diferente do valor original.

Com este método, não iremos conseguir descobrir o valor original (de entrada). Sendo assim, este método irá ser bastante importante pois, a partir do mesmo é possível garantir a segurança das chaves dos clientes.

Comparando o hash que saiu (recovered_digest) ao valor de hash inicial/inserido (digest), a pessoa pode aperceber-se da segurança dos dados, porque irá obter resultados idênticos.

2.3 Explicação do código cifradecifra.py

```
import os
from Crypto.Cipher import AES

iterations = 1000
engines = []

# Use index number 12, create a 128-bit array with it
index = 12
_128_bit_padded_index = bytes("%16d" % (index), 'utf8')

# Use the 128-bit array as the input to the multiple ciphering
data = _128_bit_padded_index
engines = []

for i in range(iterations):
    key = os.urandom(16)
    engines.append( AES.new( key, AES.MODE_ECB ) )
    data = engines[i].encrypt( data )

# Decipher in the opposite order
for i in range(iterations - 1, -1, -1):
    data = engines[i].decrypt( data )

recovered_index = int(str(data, 'utf8'))

if recovered_index == index:
    print( 'Success!' )
else:
    print( 'Failure, %d is different from %d' % (index, recovered_index) )
```

Figure 7: Código cifra/decifra

De forma a manter a segurança das chaves, iremos usar criptografia.

Para tal, iremos buscar as chaves no ciclo for e o método `.urandom` irá servir para gerar uma sequência de bytes (neste caso, 16).

A variável criptografada (`data`) agora terá o valor da mensagem criptografada em bytes.

Para descriptografar a mensagem iremos precisar da mesma chave e da mensagem criptografada (que ainda se encontra em bytes - `data`).

A variável descriptografada deverá ter o mesmo valor da mensagem original e por isso é que está feito o if (por questões de segurança).

2.4 Explicação do código tcp

```
import socket
import select

s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.bind(("localhost",1234))
s.listen()

clients = []

while True:
    try:
        available = select.select( [ s ] + clients, [], [] )[0]
    except ValueError:
        # Sockets may have been closed, check for that
        for c in clients:
            if c.fileno() == -1: # closed
                clients.remove( c )
        continue # Reiterate select

    for c in available:
        # New client?
        if c is s:
            new, addr = s.accept()
            clients.append( new )

        # Or a client message/disconnect?
        else:
            # See if client sent a message
            if len(c.recv( 1, socket.MSG_PEEK )) != 0:
                # Handle the new message received in socket c
                print(f"Mensagem recebida: {c.recv(1024)}")
            # or just disconnected
            else:
                # You may need to perform some internal cleanup of your data structures here
                clients.remove( c )
                c.close()
                break # Reiterate select
```

Figure 8: Código tcp

Em primeiro, é necessário criar um socket.

O servidor irá ter um método `bind()` que o liga a um IP e portas para que ele possa "ouvir" os pedidos recebidos.

O server também terá um método `listen()` que coloca o servidor no modo de "escuta". Isto permite que o servidor vá receber/ouvir os pedidos de entrada.

Por último, o servidor possui um método `accept()` e `close()`. O método `accept` aceita e inicia uma conexão com o cliente e o método `close` fecha a conexão com o cliente.

Se o cliente estiver "disponível" (`available`) iremos selecioná-lo e verificar se é novo ou se já enviou alguma mensagem (poderá ser um pedido para desconectar).

2.5 Explicação do Algoritmo do server_tcp

```
def main():
    # Validate the program parameters
    if len(sys.argv)!=2:
        print("ERROR: Invalid parameters <server port> <optional server ip>")
        sys.exit(1)
    if len(sys.argv)==3:
        ip = str(sys.argv[2])
    else:
        ip = "localhost"

    # Set the server's TCP address from the command args

    try: #Verificar se o inserido é valido
        address = ( ip , int(sys.argv[1]) )
        if int(sys.argv[1]) < 999:
            raise ValueError("Server port must be a number >0 with at least 4 digits")
    except ValueError as erro:
        print("ERROR: " + str(erro))
        sys.exit(1)
```

Figure 9: Função main do server

No máximo, pode-se inserir como argumentos o port do server e, opcionalmente, o server ip. Se não for optado por se dizer o server ip então será assumido como “localhost”.

```
s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
s.bind( address )
s.listen()
```

Figure 10: Sockets

O servidor que irá receber as mensagens do client.sockets irá enviar dados através da rede. Assim, o socket irá receber a conexão onde passam dois argumentos.

AF_INET refere-se à família de endereços IPv4 (figura 10).

SOCK_STREAM significa que é um socket TCP (figura 10).

O código apresentado na figura 11 irá iniciar um novo processo. Irá verificar se o cliente é ou não repetido, ou seja, irá verificar se já fez ou não um processo e, por isso, verificar se já tem uma "chave" associada, ou não.

```
def new_process( process_id, sock ):
    if process_id in procs.keys():
        return { 'error':'Sorting process already exists' }
    else:
        procs[process_id] = { 'endpoint': sock, 'ids': {} }
        return { 'error':'' }
```

Figure 11: Função new_process

```
def new_client( process_id, client_id, sock ):
    if process_id in procs:
        proc = procs[process_id] # Selected sorting process

        if client_id in proc['ids']: # Client already belongs to it?
            return { 'error':'Client already registered in sorting process' }
        else: # New client
            proc['ids'][client_id] = { 'endpoint': sock }
            return { 'error':'' }

    else:
        return { 'error':'Sorting process no found' }
```

Figure 12: Função new_client

A função new_client serve para incluir um novo cliente no programa.

No caso de o cliente já tiver realizado a sua "candidatura", o programa irá fazer o retorno de uma mensagem de erro a avisar que já está registado (if process id).

Se não estiver registado, irá ser usado um socket para receber as informações (os argumentos que entram na função são: **process_id** e **client_id**).

```
def list_clients( process_id, sock ):
    if process_id in procs:
        proc = procs[process_id] # Selected sorting process

        ids = []
        for i in proc['ids']:
            ids.append( i )

        return { 'ids':ids, 'error':'' }
    else:
        return { 'error':'Sorting process no found' }
```

Figure 13: Função list_clients

Na função `list _clients`, tal como o nome indica, iremos listar os clientes que já existem.

Isso é possível, uma vez que ao usar os `process _id` dos clientes, iremos realizar um ciclo **for** usando todos os ids que existem e fazendo a impressão (`print`) dos mesmos.

```
def run_sorting( proc ):

    def getList(dict): #faz retorno da lista que tem as chaves
        return list(dict.keys()) #vai retornar a lista das chaves do dicionario

    clients_lista = getList(proc['ids'])
    print(clients_lista) #Debug
    engines = [] #a lista das keys
    lista = [] #a lista que vai para os clientes (vai conter as keys)

    clients_lista = getList(proc['ids'])
    print(clients_lista) #Debug

    #codificar / decodificar
    for message in range(len(clients_lista)): #a mensagem funciona como uma index associado a cada cliente
        message_bytes = message.encode('utf8')
        base64_bytes = base64.b64encode(message_bytes)
        utf = base64.decode('utf8') # codificar para utf para que possa ser enviado para a função sendrecv
        lista.append(utf)

    assert len(lista) == len(clients_lista), "A lista passada aos clientes tem de conter o mesmo numero de clientes" #condição necessaria
    dict = {"list": lista, "engines": engines} # dicionario (N clientes) que contem as chaves que vao ser usadas

    #criar um ciclo for para fazer com que o dicionario passar por todos os clientes
    for i in clients_lista:
        socket_client = proc['ids'][i]['endpoint'] #Vai conter a informação dos clientes
        NDict = sendrecv_dict(socket_client, dict)
        dict.update(NDict) #faz um update do dicionario que vai ser enviado

    print(dict)
```

Figure 14: Função `run_sorting`

Esta função serve para o processo de ordenação. Em primeiro, fazemos uma lista que irá buscar as chaves contidas no dicionário (função `keys()`).

Depois fazemos um ciclo `for` (o `int message` funciona como um índice de valores associado a cada cliente) para fazer o processo de decodificação de forma a que a informação possa ser enviada para a função `sendrecv`.

Fazemos um “`assert`” para verificar se não existe nenhuma incoerência em relação ao número de clientes.

Quando verificado, o dicionário irá conter a lista e as chaves que irão ser usadas pelos clientes.

A seguir, o dicionário irá passar pelos clientes para obter (`recv`) as informações/address dos mesmos. Consequentemente, irá atualizar o dicionário anterior (o que continha a lista e as chaves).

2.6 Explicação do Algoritmo do client_tcp

```
def main():
    # Validate the program parameters
    if len(sys.argv)<4 or len(sys.argv)>5:
        print("ERROR: Parametros Inválidos <process> <id> <server port> <optional server ip>")
        sys.exit(1)

    # Set the process_id and the client_id from the command args
    process_id = sys.argv[1]
    client_id = sys.argv[2]

    try: #Verificar se o inserido é valido
        server_port = int(sys.argv[3])
        if int(sys.argv[3]) < 0:
            raise ValueError("Server port must be a number >0 ")
    except ValueError as erro:
        print("ERROR: " + str(erro))
        sys.exit(1)

    # If the program has 5 arguments, make the last one the server's ip
    if len(sys.argv) == 5:
        ip = str(sys.argv[4])
    else:
        ip = "localhost"

    # Set the server's TCP address from the command args
    address = (ip , server_port)

    s = socket.socket( socket.AF_INET, socket.SOCK_STREAM )
    s.connect( address )

    if client_id == '_':
        dump_csv ( manager( s, process_id ) )
```

Figure 15: Função main

O **sys.Argv** contém os argumentos escritos no terminal. Os argumentos começam a contar a partir do 0.

O **process id** é o segundo argumento que é passado no terminal (sendo o primeiro rali/corrida, por exemplo) logo o process_id é sys.argv[1] (o identificador especial definido é o “_”).

O **cliente id** é o terceiro argumento que passamos no terminal (logo args[2]).

O quarto argumento (args[3]) passado no terminal especifica o **porto do servidor** (TCP).

Faz-se um **if** no caso do cliente optar por colocar o **endereço IPv4** (que, neste caso, é opcional) e sendo assim passam a existir 5 argumentos sendo o último de todos eles o ip do servidor. Se não existir esta especificação, o programa assume que o cliente esteja a usar um servidor na mesma maquina onde se encontra o cliente, ou seja, **localhost**.

Por estas razões o comprimento (len) dos argumentos só podem variar entre 4 e 5. Se forem inseridos argumentos incorretos irá aparecer uma mensagem de erro como é demonstrado na imagem.

O TCP deverá ser um valor inteiro positivo por isso realizámos um **Try/except** no caso do valor inserido ser inferior a 0.

```
def client( server, proc, client ):
    request = {'op': 'ADD', 'proc': proc, 'id': client }
    resp = sendrecv_dict( server, request )

    if resp['error'] != '':
        print( 'Server error: ' + resp['error'] )
        sys.exit( 2 )

    # Wait for server orders and show the ordering outcome at the end
    dic = rcv_dict(server) # Vai receber um dicionario de 1 a N (depende do numero de participantes)
    print(dic)
    Lst = dic['lista'] # Vamos receber o que esta no dicionario

    engines = dic['engines'] # vai receber as chaves usadas (pelos participantes)

    key = os.urandom(16) # Vamos usar o método .urandom para gerar uma sequencia de bytes adequada para criptografar a lista
    engines.append(AES.new(key, AES.MODE_ECB))
```

Figure 16: Função Client

```
random.shuffle(Lst) #A lista vai ser aleatoria

for i in range(len(engines)):
    encodifica = base64.b64encode(key)
    str = encodifica.decode('utf8')
    engines[i] = str
    print(i)

dic = {"list": Lst, "engines": engines} #Vai fazer um update do dicionario com as chaves
send_dict(server, dic) #Vai enviar o dicionario atualizado (anterior)

dic = rcv_dict(server)
N=random.randint(0, len(dic['list'])) #Vai escolher um numero aleatorio entre 0 e o numero de clientes(tamanho da lista)
C = base64.b64decode(dic['list'][N]) #O elemento cifrado que cada cliente vai retirar posteriormente
K = base64.b64decode(dic['engines'])
#ci?
#ki?

#bit commitment
hash_f = SHA256.new()
hash_f.update(K)
hash_f.update(C)
```

Figure 17: Continuação da Função Client

Nesta função, pegamos nas informações contidas no dicionário(que são recebidas do server) e descodificamo-las (tanto a chave como as listas dos

clientes - N clientes).

A lista dos clientes terá que ser aleatória, por isso fazemos o random shuffle.

Depois de feito um update do dicionário (decifrado + lista aleatória) voltamos a enviar o dicionário para o server.

De seguida, escolhemos um número aleatório (N) de forma a escolher um cliente qualquer (de forma isenta) para escolher a sua chave.

Cada cliente, irá escolher um elemento cifrado (C).

De seguida, iremos enviar a chave simétrica (K) e o elemento cifrado (C) à função hash para essas informações serem guardadas de forma segura.

Passando ao código seguinte.

```
def dump_csv(data_dict):  
    # Dump a CSV to a file from data received from the server  
    csv_file = "Resultados.csv"  
    try:  
        with open(csv_file, 'w') as csvfile:  
            writer = csv.DictWriter(csvfile, delimiter=";")  
            writer.writeheader()  
            for data in dict_data:  
                writer.writerow(data)  
    except IOError:  
        print("I/O error")
```

Figure 18: Função dump_csv

Esta função serve para descarregar um csv para um ficheiro, a partir dos dados recebidos pelo servidor.

O método dict writer está contido no módulo csv. Este facto requer o nome do ficheiro csv (que está guardado como "resultados").

O código writeheader() grava a primeira linha no arquivo csv como nomes de campo. Consequentemente, o loop irá gravar cada linha.

Desta forma, iremos aceder aos valores do csv.

3 Análise dos Resultados Obtidos

Não foi possível a completa análise dos resultados obtidos, uma vez que o trabalho não foi concluído na sua totalidade.

4 Contribuição dos Autores

Para a realização deste trabalho de aprofundamento, a MO desenvolveu o código na sua maioria, fazendo a respetiva pesquisa, recolhendo o maior número de elementos que contribuíssem para o desenrolar do trabalho. A MS trabalhou na estrutura e apresentação do relatório, com a implementação de textos explicativos do código e respetivos exemplos nas imagens apresentadas acerca de cada tópico.

Assim, cada um dos intervenientes contribuiu em igual percentagem (50%) para os objetivos propostos neste trabalho.

References

- [1] <https://stackabuse.com/encoding-and-decoding-base64-strings-in-python/>
- [2] <https://docs.python.org/3/howto/unicode.html#converting-between-file-encodings>
- [3] <https://wiki.python.org.br/SocketBasico>
- [4] <http://excript.com/python/depuracao-pycharm-python.html>
- [5] <https://www.geeksforgeeks.org/python-hash-method/>
- [6] http://www.macoratti.net/vbn_cah1.htm
- [7] <https://pythonhelp.wordpress.com/tag/hash/>

Acrónimos

API Interface de Programação de Aplicações

IP Protocolo da Internet

TCP Protocolo de Controlo de Transmissão