Trabalho de aprofundamento 2

Universidade de Aveiro

Luís Couto

Trabalho de aprofundamento 2

DETI

Universidade de Aveiro

Luís Couto
(89078) luiscouto10@ua.pt
labi2019-ap2-g19

26/04/2019

Resumo

Vivemos no fenómeno da era digital, sendo cada vez mais fácil obter informação sobre qualquer tópico. Uma das ferramentas que impulsionou este fenómeno foi a *internet*.
Acessível à escala mundial, o seu acesso está disponível na forma de vários pacotes, fornecidos por várias operadoras, a vários preços e cada um com as suas características.

Este trabalho tem como objetivo desenvolver um cliente que tenha a capacidade de testar uma das características mais importantes anunciadas nos contratos de serviço de *internet*, que é a velocidade de acesso.
Neste relatório falarei do algoritmo que desenvolvi para o cliente e explicarei a maneira e o porquê da implementação desse algoritmo. Irei falar de cada função escrita, assim como apresentarei testes, bem como imagens, que demonstram o funcionamento correto do código desenvolvido. No final irei analisar os resultados e tirar conclusões.

No final deste trabalho, será possível entender como o cliente funciona, usar esse cliente para ter uma noção da largura de banda e da latência do acesso à *internet* usado e entender como a distância e o uso de servidores, muitas vezes de outros países, influênciam esse acesso.

Conteúdo

Lista de Figuras

Acrónimos

DETI - Departamento de Electrónica, Telecomunicações e Informática

LI - Laboratórios de Informática

RSA - Rivest-Shamir-Adleman

Capítulo 1

Introdução

1.1 Motivação

No âmbito da avaliação da cadeira de LI, foi proposto efetuar um trabalho de aprofundamento, a realizar em *Python*, com o objetivo de criar um cliente para os servidores pertencentes ao serviço ***speedtest***[1].

Muitas das vezes, a velocidade de acesso real é mais baixa do que a anunciada no contrato do serviço pago e embora existam alguns fatores que influenciem esta velocidade, o cliente merece ter o que paga. Como este trabalho visa a criação de uma aplicacão que permita testar certas caracteristícas da velocidade de acesso, tais como a latência e a largura de banda, de uma maneira rápida e de uso fácil, podemos dizer que este trabalho tem uma certa importância.

1.2 Conteúdo disponíbilizado

Quanto ao trabalho, este tem o seu conteúdo distribuído por 2 pastas, sendo que cada uma contém os seguintes ficheiros:

1. codigo
   (a) client.py
   (b) test_unitario_funcoes.py
   (c) servers.json
2. relatorio

(a) TrabalhoAP2.pdf

(b) Todas as imagens usadas

(c) Todos os ficheiros fonte necessários

Quando a este relatório, está dividido em 5 capítulos. Depois desta introdução, no Capítulo 2 falo da implementação do algoritmo e as razões pelas quais decidi o implementar da maneira que implementei. No Capítulo 3 vou esclarecer como implementei esse algoritmo, apresentando imagens do código acompanhadas de explicações.
No Capítulo 4 falo dos testes unitários que realizei e o resultado deles, sendo que por fim, no Capítulo 5 apresento uma conclusão.

1.3 Instalações necessárias para o funcionamento correto do código

Este código faz uso de vários *frameworks*, pacotes e métodos que não fazem parte da biblioteca *built-in* do *Python*[2], pelo que é absolutamente necessário que o utilizador tenha instalado os seguintes pacotes:

1. ***PyCryptodome*** [3]
2. ***pytest*** [4]

Capítulo 2

Algoritmo

2.1 Estruturação do Algoritmo

Para a implementação do algoritmo, tendo em conta que o trabalho tinha vários requisitos que se interligavam e funcionavam em conjunto, decidi fazer funções que realizam todas as verificações de argumentos e variáveis, sinteses e cálculos relacionados com a velocidade de internet.

O código da maioria das funções está dentro de um *Try...Except* [5] sendo que cada um com tem as excepções mais comum quando se tenta correr o tipo de código em questão.
No final existe uma função *main* onde todos os comandos e funções são usados e interligados, de modo a produzir o resultado final.

2.2 Funcionamento do Algoritmo

Este algoritmo tem como propósito receber um input do utilizador, na forma **python3 client.py interval num [country or id]**.

O argumento **interval** irá corresponder ao tempo que decorre entre cada teste, o argumento **num** ao número de testes a realizar. O terceiro argumento pode ser **country** que é o país hospedeiro do servidor, ou **id** que é o id do servidor em questão.

Após este input, o algoritmo verifica cada argumento introduzido, verificando se cumpre certos requisitos. Se os argumentos tiverem verificação positiva,o programa continua e vamos ler o ficheiro ***servers.json***, de modo a encontrar um servidor que tenha o id igual ao introduzido ou que seja hospedado pelo país introduzido.

Se o servidor não for encontrado, o programa acaba, caso contrário o algoritmo executa 3 comandos:

1. A função ***sendHi***, que dá return à versão do software, à data e à hora do servidor
2. A função ***sendPing***, que dá return à latência da comunicação
3. A função ***sendDownload***, que dá return à largura de banda

Estes valores, em conjunto com o número do teste, a data no formato ISO e a síntese dos campos anteriores vão formar cada linha do ficheiro ***report.csv***.

O algoritmo executa estas instruções ***num*** vezes, com um intervalo de ***interval*** segundos entre cada teste, escrevendo cada linha do ficheiro ***report.csv*** à medida que realiza cada teste.

Após este ficheiro estar completamente escrito, o algoritmo gera um ficheiro ***key.priv***, que contém uma chave privada RSA. É feita a síntese do ficheiro ***report.csv*** e, após isso, é lido o ficheiro ***key.priv*** e usada a chave nele contida para gerar uma assinatura do ficheiro ***report.csv***.

Finalmente é gerado o ficheiro ***report.sig***, que contém essa assinatura, acabando assim o programa.

Capítulo 3

Implementação do Algoritmo - client.py

Neste capítulo vai ser explicada a implementação do algoritmo, esclarecendo o seu funcionamento. Tendo em conta que abordei o trabalho fazendo funções e depois uma função *main*, vou explicar cada uma delas.

3.1 error_list

Apesar de não ser uma função, este bloco de código é muito importante pois serve para mapear os erros que as funções podem devolver. Cada algarismo corresponde a uma mensagem de erro, sendo que as funções podem dar *return* a um esses algarismos.
O algarismo 0 não está mapeado, porém é um valor de *return* que significa que não houve erro e que o código vai continuar a ser executado.

```
# MAPEAR OS ERROS POSSIVEIS
error_list = {
        1: "Escreva no formato: python3 client.py interval num [country or id]",
        2: "Valor do argumento deve ser inteiro positivo",
        3: "Argumento do tipo errado, deve ser do tipo integer",
        4: "Country não existe na lista",
        5: "ID não existe na lista"
    }
#------------------------------------------------------------------------------
```

Figura 3.1: Error_list

3.2 Função validateArgs

Esta função tem como argumento o número total de argumentos introduzidos pelo utilizador e serve para validar o número de argumentos.
Se forem introduzidos menos do que 4 argumentos (nome do ficheiro conta como 1 argumento), a função dá *return* de 1, que é um erro mapeado. Se forem introduzidos 4 ou mais argumentos, a função dá *return* de 0, que faz com que o código continue a ser executado.

```
# FUNÇÃO PARA VALIDAR O NÚMERO DE ARGUMENTOS
def validateArgs(totalArgs):
    if (totalArgs < 4):
        return 1
    return 0
#-------------------------------------------------------------
```

Figura 3.2: Função validateArgs

3.3 Função isInteger

Esta função tem como argumento um valor de qualquer tipo e verifica se esse valor é inteiro positivo.
Dentro to bloco *Try* tentamos converter o valor para inteiro. Se for possível, verificamos se é maior ou igual a 0 (inteiro positivo) e, caso isto se verifique, o valor é válido, função dá *return* de 0. Caso não se verifique, a função dá *return* de 2, que é um erro mapeado.
Se não for possível converter o número para inteiro, a função levanta uma excepção do tipo *ValueError* e dá *return* de 3, que é um erro mapeado.

```
# FUNÇÃO PARA VERIFICAR SE O INPUT É UM INTEIRO POSITIVO
def isInteger(value):
    try:
        int(value)
        if(value <= "0"):
            return 2
        else:
            return 0
    except ValueError:
        return 3
#-------------------------------------------------------------
```

Figura 3.3: Função isInteger

3.4 Função validateIDExists

Esta função tem 2 argumentos que são o ID a validar e o ficheiro em qual queremos procurar esse ID. Vamos percorrer cada servidor do ficheiro e caso o ID de um desses servidores seja igual ao argumento ID que a função recebe, temos *return* de 0 e o código continua a ser executado.
Caso nenhum ID corresponda, a função dá *return* de 5, que é um erro mapeado.

```
# FUNÇÃO PARA VERIFICAR SE EXISTE ALGUM SERVIDOR COM ID == INPUT [ID]
def validateIDExist(id, file):
    for x in file["servers"]:
        if (int(x["id"])==int(id)):
            return 0
    return 5
#---------------------------------------------------------------------------
```

Figura 3.4: Função validateIDExists

3.5 Função validateCountryExists

Esta função tem 2 argumentos que são o país a validar e o ficheiro em qual queremos procurar esse país. Vamos percorrer cada servidor do ficheiro e caso o país de um desses servidores seja igual ao argumento country que a função recebe, temos *return* de 0 e o código continua a ser executado.
Caso nenhum país corresponda, a função dá *return* de 5, que é um erro mapeado.

```
# FUNÇÃO VERIFICAR SE EXISTE ALGUM SERVIDOR COM COUNTRY == INPUT [COUNTRY]
def validateCountryExist(country, file):
    for x in file["servers"]:
        if (x["country"]==country):
            return 0
    return 4
#---------------------------------------------------------------------------
```

Figura 3.5: Função validateCountryExists

3.6 Função printError

Esta função tem 1 argumento que é número do erro. Caso seja um erro mapeado, a função dá *return* da respetiva mensagem associada ao número do erro.
Caso não seja um erro mapeado, a função dá *return* de "Unknown Error".

```
# FUNÇÃO IMPRIMIR UMA MENSAGEM COM O RESPETIVO ERRO
def printError(error_num):
    return error_list.get(error_num, "Unknown Error")
#-------------------------------------------------------------
```

Figura 3.6: Função printError

3.7 Função getHostById

Esta função tem 2 argumentos que são o ID e o ficheiro em qual queremos procurar esse ID. Vamos percorrer cada servidor do ficheiro e caso o ID de um desses servidores seja igual ao argumento ID que a função recebe, a função dá *return* do *host* desse servidor.

```
# FUNÇÃO PARA OBTER O HOST ATRAVES DO INPUT [ID]
def getHostByID(id, file):
    for x in file["servers"]:
        if (int(x["id"])==int(id)):
            return x["host"]
#-------------------------------------------------------------
```

Figura 3.7: Função getHostByID

3.8 Função getHostByCountry

Esta função tem 2 argumentos que são o país a e o ficheiro em qual queremos procurar esse país. Vamos percorrer cada servidor e caso o país desse servidor seja igual ao argumento country que a função recebe, vamos incrementar 1 à variável count (que é o número de vezes que esse país aparece).
Depois vamos gerar um número aleatório de 1 até ao número de vezes que o país aparece. Corremos todos os servidores e mais uma vez vemos quantas vezes esse país aparece, depois quando o índex for igual ao número gerado, damos *return* ao *host* que tenha esse índex.

```
# FUNÇÃO PARA OBTER O HOST ATRAVES DO INPUT [COUNTRY]
def getHostByCountry(country,file):
    count=0;
    index=0;
    for x in file["servers"]:
        if(x["country"]==country):
            count=count+1;

    rnd = randint(1, count+1);
    for x in file["servers"]:
        if (x["country"]==country):
            index = index+1
            if (index == rnd):
                return x["host"]
#---------------------------------------------------------------------------
```

Figura 3.8: Função getHostByCountry

3.9 Função sendHi

Esta função tem 2 argumentos que são a *host* e o *port* do servidor que queremos testar. O código esta dentro de um bloco *Try...Except* que nos permite apanhar as excepções mais comuns neste tipo de código.
Caso o código no bloco *Try* corra sem levantar excepções, a função dá *return* dos dados recebidos, descodificados. Caso ocorra uma excepção, a função dá *return* de uma mensagem com o respetivo erro.

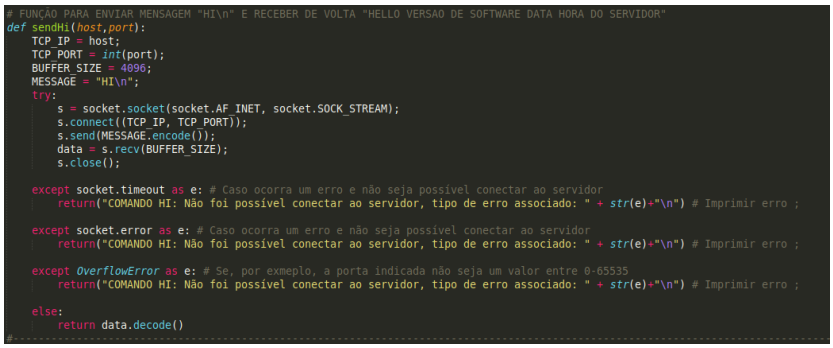

Figura 3.9: Função sendHi

3.10 Função sendPing

Esta função tem 3 argumentos que são a *host*, o *port* do servidor que queremos testar e o *timestamp* do momento em que mandamos a mensagem.

Dentro do bloco *Try*, temos todo o código que queremos correr. Iniciamos a variável tmp_lat_sum fora do ciclo for pois necessitamos de incrementar valores a ela cada vez que o ciclo faça uma iteração. Obtemos a latência de cada iteração e vamos somar todos esses valores.
Fora do ciclo *for* dividimos esse valor de latência pelo número de iterações, neste caso 10, obtendo assim o valor médio de 10 transações PING/PONG que é a latência.

Caso ocorra uma excepção, a função dá print do erro que ocorreu e dá *return* de -1.

```
# FUNÇÃO PARA CALCULAR A LATÊNCIA
def sendPing(host,port,timestamp):
    TCP_IP = host;
    TCP_PORT = int(port);
    BUFFER_SIZE = 4096;
    MESSAGE = "PING "+str(timestamp)+"\n";

    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM);
        s.connect((TCP_IP, TCP_PORT))
        tmp_lat_sum = 0

        for i in range(1,11):
            time_start = time.time()*1000;

            s.send(MESSAGE.encode('utf-8'));
            data = s.recv(BUFFER_SIZE);

            time_end = time.time()*1000;

            tmp_lat = abs(time_end - time_start);

            tmp_lat_sum += tmp_lat;

        s.close();

        v_latencia = (tmp_lat_sum)/10

        latencia = (math.ceil(v_latencia*1000)/1000)
```

Figura 3.10: Função sendPing - Bloco Try

```
except socket.timeout as e:
    print("COMANDO DOWNLOAD: Não foi possível conectar ao servidor, tipo de erro associado: " + str(e))
    return -1;

except socket.error as e:
    print("COMANDO PING: Não foi possível conectar ao servidor, tipo de erro associado: " + str(e))
    return -1;

except OverflowError as e:
    print("COMANDO PING: Não foi possível conectar ao servidor, tipo de erro associado: " + str(e))
    return -1;

else:
    return latencia
```

Figura 3.11: Função sendPing - Bloco das Excepções

3.11 Função sendDownload

Esta função tem 2 argumentos que são a *host* e o *port* do servidor que queremos testar.

O tamanho do *download* vai ser um inteiro aleatório entre 10000000(10MB) e 100000000(100MB). Fazemos um ciclo *while* que corre enquanto não tiverem passados 10 segundos.
Dentro desse ciclo temos outro ciclo *while* que corre enquanto não tiverem sido feitos um *download* de 1MB. Dentro deste ciclo vamos ter uma variável para contar o tempo, que vai actualizar até ter sido feito um *download* de 1MB. Após isso temos o tempo em que começa, que se mantém constante. Incrementamos o número de octetos que são baixados á medida que o ciclo *while* corre.

Calculamos a largura de banda em *Megabytes* por segundo e convertemos para *Megabits* por segundo. Caso ocorra uma excepção, a função dá print do erro que ocorreu e dá *return* de 0.

```
# FUNÇÃO PARA CALCULAR A LARGURA DE BANDA
def sendDownload(host,port):
    size = randint(10000000,100000000)
    TCP_PORT = int(port);
    BUFFER_SIZE = 4096;
    MESSAGE = "DOWNLOAD "+str(size)+"\n";

    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((TCP_IP, TCP_PORT))

        tempo_inicial = time.time()
        tempo_final = tempo_inicial + 10

        banda_total = 0
        banda_inicial = 0

        while(time.time() <= tempo_final):

            s.send(MESSAGE.encode("utf-8"));
            data = s.recv(BUFFER_SIZE)

            while (banda_total <= 1000000):
                tempo_inicial_banda = time.time()
                banda_inicial += len(data.decode("utf-8"))
                break


            if (size==0):
                return 0;
                break

            if not data:
                return 0;
```

Figura 3.12: Função sendDownload

```
        banda_total += len(data.decode("utf-8")) # Como fazemos download por pacotes, incrementamos cada pacote. Damos decode para puder usar a funçao len
    s.close()

    banda = banda_total - banda_inicial # Numero de octetos recebidos após ter sido feito download de 1 MB

    tempo_total = tempo_final - tempo_inicial_banda # Calcular o tempo decorrido após ter sido feito  download de 1 MB

    banda_MB = banda * 0.000001; # Calcular o número de Megabytes(MB) que foram descarregados

    taxa_banda_MB = banda_MB / tempo_total # Obter taxa de banda em Megabytes por Segundo

    v_taxa_banda = (taxa_banda_MB)*8 #  Converter MB/s para Mbps (Megabytes por Segundo para Megabits por Segundo)

    taxa_banda = (math.ceil(v_taxa_banda*1000)/1000) # Arredondar para 3 casas decimais

except socket.timeout as e:
    print("COMANDO DOWNLOAD: Não foi possível conectar ao servidor, tipo de erro associado: " + str(e))
    return 0; # Taxa de banda toma o valor 0

except socket.error as e: # Caso ocorra um erro e não seja possível conectar ao servidor
    print("COMANDO DOWNLOAD: Não foi possível conectar ao servidor, tipo de erro associado: " + str(e))
    return 0; # Taxa de banda toma o valor 0

except OverflowError as e: # Caso a porta indicada não seja um valor entre 0-65535
    print("COMANDO DOWNLOAD: Não foi possível conectar ao servidor --> tipo de erro associado: " + str(e))
    return 0; # Taxa de banda toma o valor 0

else:
    return taxa_banda
#-----------------------------------------------------------------------------------------------------------------------
```

Figura 3.13: Função sendDownload

3.12 Função getSintese

Esta função tem 1 argumento que é o campo que queremos fazer a síntese. Nesta função usamos a SHA3_512, que é das mais recentes mas também das mais seguras. Caso ocorra uma excepção, a função da print do erro que ocorreu e o programa dá *exit*.

```
# FUNÇÃO PARA CALCULAR A SÍNTESE DOS CAMPOS (JÁ CONCATENADOS E SEM SEPARADORES)
def getSintese(check):
    try:
        chave = SHA3_512.new() # Vou usar SHA3_512 pois é das mais seguras
        chave.update(check.encode('utf-8'))
        sintese = chave.hexdigest()

    except TypeError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except ValueError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except AttributeError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except NameError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    else:
        return sintese
#-----------------------------------------------------------------------------------------------------------------------
```

Figura 3.14: Função getSintese

3.13 Função getFileSintese

Esta função tem 1 argumento que é o ficheiro que queremos fazer a síntese. O buffer é cada linha do ficheiro e enquanto o ficheiro tiver linhas, vamos percorrer e fazer a síntese.
Caso ocorra uma excepção, a função da print do erro que ocorreu e o programa dá *exit*.

```
# FUNÇÃO PARA CALCULAR A SÍNTESE DE UM FICHEIRO COMPLETO
def getFileSintese(filename):
    try:
        chave  = SHA3_512.new();
        ficheiro = open( filename, "rb" )
        buffer = ficheiro.readline()
        chave.update(buffer)

        while len(buffer) > 0:
            buffer = ficheiro.readline()

            chave.update(buffer)

        sintese = chave.hexdigest()

    except OSError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except TypeError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except ValueError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except AttributeError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    except NameError as e:
        print("Erro na linha {},".format(sys.exc_info()[-1].tb_lineno) + " do tipo: " + str(e))
        sys.exit(1)

    else:
        return sintese
#-------------------------------------------------------------------------------------------------------------------
```

Figura 3.15: Função getFileSintese

3.14 Função main

Esta função é a principal, que utiliza todas as funções anteriores e produz o resultado final. Devido ao tamanho, não vão haver imagens porém o código está todo comentado.

Começamos por validar todos os argumentos usando as funções necessárias, depois realizamos os testes usando um *for loop*, sendo que por casa iteração, são calculadas as caracteristícas de velocidade de acesso á internet. No fim do ciclo *for*, geramos um ficheiro chamado ***key.priv***, que contém uma chave privada RSA. É feita a síntese do ficheiro ***report.csv*** e depois é lido o ficheiro ***key.priv***. Usando a chave nele contida, geramos um ficheiro com a assinatura do ficheiro ***report.csv***.

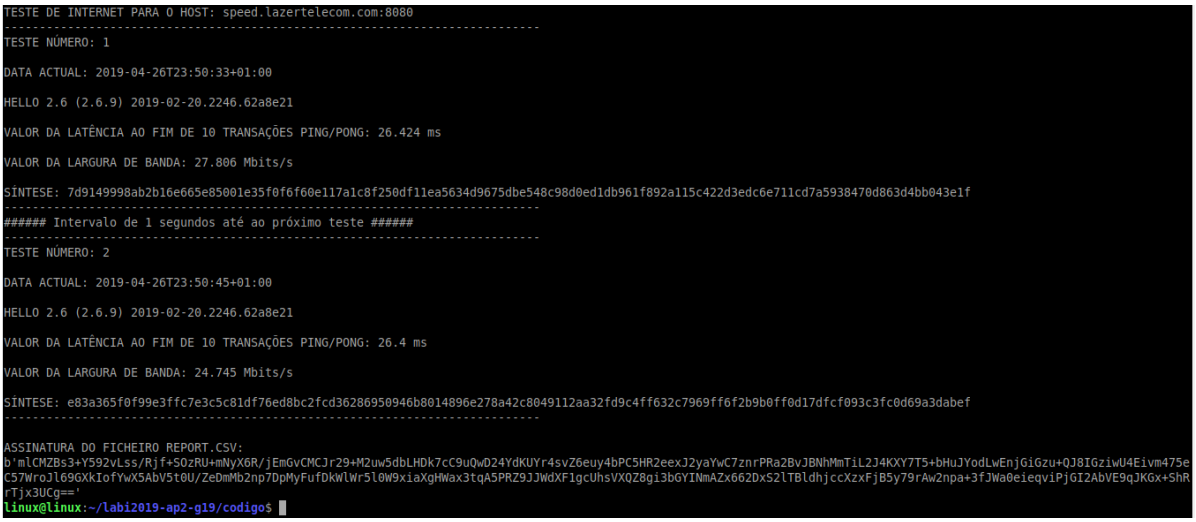

Figura 3.16: Função main após ser corrida

Capítulo 4

Implementação de testes unitários - test_unitario_funcoes.py

Neste capítulo vai ser uma breve exposição dos testes unitários das funções. As funções que não estão testadas são as ***sendHi***, ***sendPing*** e ***sendDownload*** pois os seus resultados dependem de muitas variáveis que não podem ser controladas.

O código está todo comentado, pelo que a sua leitura é fácil. Os testes são realizados de modo a verificar que as funções dão os resultados esperados, que dão, como podemos ver pela seguinte imagem.

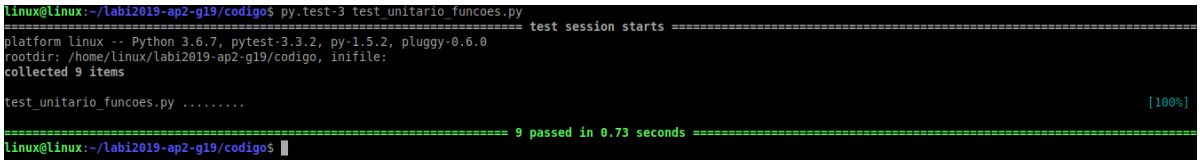

Figura 4.1: Testes das funções

Capítulo 5

Conclusão:

Vivemos num mundo dominado pela *internet*, é uma ferramenta que usamos todos os dias, quer seja para trabalho ou para lazer.

Com este trabalho, é possível agora ter uma ideia de como a velocidade de acesso à *internet* é definida e quais os vários aspetos a ter em conta quando escolhemos um pacote de *internet*. É possível também observar, pelo uso da aplicação, que a distância e o local onde o servidor está, em relação à nossa posição, tem influência na velocidade de acesso. Espero que com este trabalho, as pessoas percebam um pouco melhor de como a *internet* funciona.

5.1 Contribuição dos autores

O aluno Luís Couto ficou encarregue da realização de todo o código e do relatório.

Tendo isto em conta, a contribuição de LC foi 100%.

Bibliografia

[1] O. LCC. (2019). Speedtest Servers, URL: https://www.speedtest.net/speedtest-servers.

[2] P. S. Foundation. (2019). The Python Standard Library, URL: https://docs.python.org/3/library/.

[3] R. T. Docs. (2019). Installation, URL: https://pycryptodome.readthedocs.io/en/latest/src/installation.html.

[4] H. Krekel e pytest-dev team. (2019). Installation and Getting Started, URL: https://docs.pytest.org/en/latest/getting-started.html.

[5] P. S. Foundation. (2019). Errors and Exceptions, URL: https://docs.python.org/3/tutorial/errors.html.