

Guess the number - AP2

Universidade de Aveiro

Diogo Matos, Miguel Gomes



Guess the number - AP2

DETI

Universidade de Aveiro

Diogo Matos, Miguel Gomes
(102848) dftm@ua.pt, (103826) mig.gom@ua.pt

30-05-2021

Resumo

Foi-nos pedido, no âmbito da unidade curricular de Laboratórios de Informática (LABI), que desenvolvêssemos um jogo do tipo guess the number, utilizando os conhecimentos da linguagem Python, sockets Tcp, Comma-separated values (CSV), testes e depuração, adquiridos nas aulas. Resumidamente o objetivo passou pela criação de dois programas, o cliente e o servidor. Estes dois comunicariam um com o outro a partir de sockets, mensagens em formato dicionário que sucintamente apresentavam toda a informação necessária para comunicar as intenções pretendidas, e que permitissem que o funcionamento do jogo. No fim era previsto ter-se criado um programa robusto, funcional e com código limpo, que permitisse uma interação cliente / servidor intuitiva e que realizasse o esperado.

Neste relatório irá ver todos os resultados dos testes e simulações realizadas aos programas antes descritos, com o objetivo de provar que funcionam como esperado.

Agradecimentos

Queríamos agradecer a todos os responsáveis pela unidade curricular (UC) de LABI, que com a proposta de realização deste trabalho, desencadearam o nosso aumento de conhecimentos gerais e específicos.

Índice

1	Introdução	1
2	Metodologia	2
3	Resultados e análise	3
3.1	Robustez da inicialização das aplicações	3
3.2	Funções essenciais	4
3.3	Simulação de uma interação cliente-servidor	11
3.4	Segurança - uso de incryptação	13
4	Análise geral	15
5	Conclusões	16

Capítulo 1

Introdução

O trabalho descrito ao longo deste relatório foi realizado no âmbito da UC de LABI. De acordo com o guião do trabalho [1] o principal objetivo é:

Criar um servidor que suporte a geração de um número inteiro aleatório (entre 0 e 100), que vamos designar por número secreto, bem como o número máximo de tentativas (entre 10 e 30) concedidas para o adivinhar. E um cliente que permita adivinhar esse número secreto. Ou seja um jogo de adivinha o número secreto.

O servidor não aceita dois clientes com a mesma identificação dando erro no caso de acontecer. O trabalho para além disso produz um ficheiro em formato CSV onde é registado o nome do cliente, o numero de tentativas máximas, o numero de tentativas feitas pelo mesmo, o numero da sorte que o servidor gerou e o resultado final. Por ultimo o jogo também pode funcionar com ou sem encriptação dos números, ficando ao critério do cliente a escolha de usar encriptação ou não. O cliente pode pedir informações do jogo logo quando inicializa e pode desistir a qualquer momento.

Neste relatório apresentamos os Resultados e análise dos testes e simulações, e explicações breves dos algoritmo conforme se vai progredindo na análise dos programas. No Capítulo 4 resumi-mos o que se retirou dos testes e simulações e, por fim, os comentários finais nas Conclusões.

Capítulo 2

Metodologia

De forma a testar o programa, dividi-mos a análise em quatro partes:

- Robustez da inicialização das aplicações.
- Funções essenciais.
- Simulação de uma interação cliente-servidor.
- Segurança - uso de encriptação.

Assim, demonstramos a devida correção do algoritmo a partir de testes unitários e depuração. Para a realização dos testes unitários foram criados pequenos programas posteriormente executados com **pytest**, esse mesmo código está presente no próximo capítulo juntamente com os resultados e a análise dos mesmos. Na simulação cliente / servidor e segurança usou-se a ferramenta python **IPython-enabled Python Debugger (ipdb)**¹, assim podendo analisar os valores das variáveis em diferentes momentos da execução do programa, possibilitando a visualização de um 'caminho' tomado até ao output final.

¹O ipdb é um módulo built-in que funciona como um console interativo, onde é possível realizar debug de código python.

Capítulo 3

Resultados e análise

Neste capítulo, subdividido em quatro secções como referido no Capítulo 2, estão presentes os resultados dos testes e simulações realizadas, seguidos pela sua devida análise. Tentámos focar este teste nos resultados finais e não tanto a uma explicação mais pormenorizada. Mesmo assim, a partir do que será apresentado a seguir, será capaz de ter uma melhor percepção do funcionamento do programa cliente e servidor, das funções que os constituem, do raciocínio por trás do algoritmo e a devida correção de todos os seus constituintes.

3.1 Robustez da inicialização das aplicações

Ao iniciar o programa no terminal temos de digitar o comando `python3 [server.py] [porto]`, por exemplo `python3 server.py 1234`, no diretório onde se encontra o programa. No exemplo, o terceiro argumento `1234` é o porto que vamos utilizar para realizar a comunicação entre servidor e cliente e tem de ser um numero inteiro positivo, na seguinte tabela vai ser mostrado vários exemplos de erros ao inicializar o servidor e o seu respectivo resultado.

Erros de inicialização	
Valor de entrada	Mensagem de erro
<code>python3 server.py ola</code>	Passe um valor de porto do tipo inteiro positivo
<code>python3 server.py 20.3</code>	Passe um valor de porto do tipo inteiro positivo
<code>python3 server.py -235</code>	Passe um valor de porto do tipo inteiro positivo
<code>python3 server.py</code>	Numero errado de argumentos (comando correto: <code>python3 server.py porto</code>)
<code>python3 server.py 1 xau</code>	Numero errado de argumentos (comando correto: <code>python3 server.py porto</code>)

Já iniciado o servidor, podemos iniciar o cliente no mesmo diretório no terminal devemos inserir `python3 [client.py] [nome do cliente] [porto] [hostname]`, o **hostname** não é obrigatório inserir, se todos os argumentos anteriores forem válidos e o utilizador não passar o **hostname** como argumento o programa inicia sem erro, assumindo o mesmo como `127.0.0.1`. Na seguinte tabela vão estar os erros que serão apanhados pelo cliente na inicialização.

Nota. O servidor sendo iniciado numa determinada porta, o cliente tem de ser iniciado na mesma porta. Na tabela para testar a inicialização do cliente, anteriormente inicializamos o servidor na porta 1234

Erros de inicialização	
Valor de entrada	Mensagem de erro
<code>python3 client.py 1234</code>	Numero de argumentos errado (python3 client.py client_id porto [máquina])
<code>python3 client.py Miguel 1234 ola</code>	Passe um valor de hostname valido (ex: "123.1.2.3")
<code>python3 client.py -235</code>	Passe um valor de porta do tipo inteiro positivo
<code>python3 client.py Miguel 123</code>	Impossivel conectar ao servidor, verifique se colocou o valor de hostname correto e o valor da porta correta
<code>python3 client.py</code>	Erro: numero de argumentos errado (python3 client.py client_id porto [máquina])
<code>python3 client.py Miguel 1234 2</code>	Impossivel conectar ao servidor, verifique se colocou o valor de hostname correto e o valor da porta correta

3.2 Funções essenciais

Antes de começar é importante referir que as funções criadas, principalmente as do servidor, como não recebem input diretamente de um ser humano, não estão desenhadas para apanhar alguns tipos de erro, como por exemplo, o número de entradas no dicionário recebido. A estrutura dos argumentos passados a estas funções são validados no programa do cliente, sendo que só parte deles dependem do input humano.

Começando, a função que permite jogar é nomeada por **guess_client**. Esta função recebe o socket do cliente e a mensagem enviada pelo mesmo como argumentos de entrada, a partir do socket acede ao seu id, usando a função **find_client_id**. Depois de verificar todas as condições necessárias, devolve um dicionário de erro ou de sucesso, que indicam se a escolha do jogador é maior, menor ou igual ao número correto, usando a função **guess_is_correct** para essa comparação. Neste teste unitário é possível comprovar o bom funcionamento das três funções anteriormente referidas, incluindo os seus mecanismos de detecção de erros. É importante apontar que o dicionário **gamers**, usado para guardar as informações do jogador, que foi usado para estes testes foi

o seguinte: {"Manel" : {"socket" : "abc", "guess" : 60, "max_attempts" : 20, "attempts" : 0, "cipher" : None}}.

```
import pytest

from server import (guess_is_correct, guess_client, find_client_id)

def test():
    assert find_client_id("abc") == "Manel"
    assert guess_client("abc", {"op": "GUESS", "number": 50, "cipher": None}) == { "op":
        "GUESS", "status": True, "result": "smaller" }
    assert guess_client("abc", {"op": "GUESS", "number": 80, "cipher": None}) == { "op":
        "GUESS", "status": True, "result": "larger" }
    assert guess_client("abc", {"op": "GUESS", "number": 60, "cipher": None}) == { "op":
        "GUESS", "status": True, "result": "equals" }
    assert guess_client("abcdef", {"op": "GUESS", "number": 60, "cipher": None}) == { "
        op": "GUESS", "status": False, "error": "Cliente inexistente" }
    assert guess_client("abc", {"op": "GUESS", "number": 105, "cipher": None}) == { "op"
        : "GUESS", "status": False, 'error': 'Valor fora dos limites (0 <= number <= 100)
        ' }
```

Figura 3.1: código do programa de teste

```
===== test session starts =====
platform linux -- Python 3.8.5, pytest-4.6.0, py-1.8.1, pluggy-0.13.0
rootdir: /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/la1/ap2/la12021-ap2-g2/client-server/testes
collected 1 item

testGuess.py . [100%]

===== 1 passed in 0.58 seconds =====
```

Figura 3.2: Resultado do teste unitário

A próxima função a testar é a **stop_client**, esta é chamada quando o cliente acerta e, assim, acaba o jogo. Esta função recebe o socket do cliente e a mensagem enviada pelo mesmo como argumentos. Inicialmente a partir da função, anteriormente testada, **find_client_id** acede ao id do cliente e em seguida depois de verificar possíveis erros como:

1. Inconsistência do numero de jogadas.
2. O valor de jogadas maximas exedido.
3. Cliente não encontrado.

devolve um dicionario de sucesso onde está presente o numero secreto.

```
import pytest

from server import (stop_client, gamers, guess_client)
from client import stop_action

def test():
    assert stop_client("abc", {"op": "STOP", "number": 60, "attempts": 20}) == { "op": "STOP", "status": True, "guess": 60}
    assert stop_action("abc", 20) == ("— Parabéns!!! Terminou o jogo em 20 tentativas —")
    assert stop_client("abcdef", {"op": "STOP", "number": 32, "attempts": 20}) == { "op": "STOP", "status": False, "error": "Cliente inexistente" }
    assert stop_client("abc", {"op": "STOP", "number": 60, "attempts": 8}) == { "op": "STOP", "status": False, "error": "Numero de jogadas inconsistente" }
    increment = guess_client("abc", {"op": "GUESS", "number": 50}) #para incrementar a contagem, para ser possivel realizar a proxima testagem
    assert stop_client("abc", {"op": "STOP", "number": 60, "attempts": 21}) == { "op": "STOP", "status": False, "error": "Excedeu o numero maximo de tentativas" }
```

Figura 3.3: Código do programa de teste

Depois da função **stop_client** ser executada a mensagem do resultado final é imprimida, esta provém da função **stop_action** que é chamada na função **run_client** do lado do cliente, quando o cliente acerta no número secreto e não ocorre qualquer erro nesse processo. A função **stop_action** imprime uma mensagem onde aparece o número de jogadas efetuadas e se o cliente venceu ou perdeu, por fim, fecha o socket do cliente e termina o programa do mesmo. No teste acima apresentado aproveitou-se também para verificar esta função, para isso dentro da função trocou-se as instruções de **print** para **return**, assim abrindo a porta à simulação decorrendo a um **assert**. Como antes referido, na função **stop_action** o programa é terminado e o socket do cliente fechado (que não foi aberto no teste), assim sendo impossível fazer qualquer teste unitário sem eliminar essas instruções, logo foram retiradas momentaneamente para os testes serem realizados. A instrução de limpeza do dicionário **gamers** também foi retirada, por razões similares às anteriores. As instruções eliminadas, mais à frente, na Seção 3.3 serão dadas a observar a funcionar.

Se observar o código de teste vê que se fez um GUESS de forma a aumentar em uma unidade o número na chave **attempts** guardado dentro do dicionário, assim o seu valor passa de 20 para 21, logo excede o **max_attempts**, possibilitando o teste do erro 3 presente na lista anterior. Por fim, foi considerado o dicionário **gamers**: {"Manel": "socket": "abc", "guess": 60, "max_attempts":

20, "attempts": 20, "cipher": None}}.

```
===== test session starts =====
platform linux -- Python 3.8.5, pytest-4.6.9, py-1.8.1, pluggy-0.13.0
rootdir: /mnt/c:/Users/usuario/Desktop/diogo/uni/repositorios/labi/ap2/labi2021-ap2-g2/client-server/testes
collected 1 item
testStop.py . [100%]
===== 1 passed in 0.50 seconds =====
```

Figura 3.4: Resultado do teste unitario

Comprova-se o funcionamento correto da função **stop_client** e, parcialmente, da **stop_action**.

As próximas funções a ser chamadas são as funções **new_client**, **quit_client** e **clean_client**. Começando pela função **new_client** esta é chamada logo no início do programa, quando o cliente decide iniciar jogo com ou sem encriptação introduzindo no terminal "y" ou "n", introduzindo uma destas duas opções o client envia um socket e uma mensagem em formato j_son com o seguinte texto {"op": "START", "client_id": sys.argv¹[1], "cipher": cipherkey_tosend}. O servidor recebe a mensagem do cliente e chama a função **new_client**, esta função é chamada sempre que o cliente envia "START" no campo "op", os argumentos da função são o socket e a resposta do cliente. O que a função faz é :

1. Verificar se o cliente já existe.
2. Guardar os valores essenciais do jogo no dicionário gamers.
3. Enviar uma mensagem ao cliente com o número de jogadas máximas

Caso o cliente já exista a função deve devolver uma mensagem de erro, caso não exista deve devolver uma mensagem ao cliente com o número máximo de jogadas . Neste teste também usamos o cliente Manel.

¹sys.argv é utilizado para aceder à lista de argumentos passados pela linha de comandos em que o número é o local onde se posiciona o argumento que queremos aceder

```

import pytest

from server import (update_file, find_client_id, new_client, quit_client, gamers,
                    clean_client)

def test():
    assert new_client("abc", {"op": "START", "client_id": "Manel", "cipher": None}) == {
        "op": "START", "status": False, "error": "cliente existente" }
    assert new_client("abcd", {"op": "START", "client_id": "Paula", "cipher": None}) ==
        { "op": "START", "status": True, "max_attempts": gamers["Paula"]["max_attempts"]
        }
    assert quit_client("abcd", {"op": "QUIT"}) == { "op": "QUIT", "status": True }
    assert quit_client("abcde", {"op": "QUIT"}) == { "op": "QUIT", "status": False, "
        error": "cliente inexistente" }
    assert clean_client("abc") == True
    assert clean_client("4515") == False

```

Figura 3.5: código do programa de teste

Depois da função **new_client** testamos a função **quit_client**, esta função é chamada sempre que o cliente enviar a seguinte mensagem {"op": "QUIT"}. A função recebe dois argumentos: o `client_sock` e a mensagem do cliente. Se o `client` já está registado no dicionário `gamers`, a função tem três operações a fazer:

1. Atualizar o ficheiro **report.csv** através da função `update_file`.
2. Eliminar o cliente do dicionário `gamers`, através da função `clean_client`.
3. Enviar uma mensagem ao cliente com a operação Quit e status **True**

Caso o cliente não esteja registado no dicionário só deve enviar uma mensagem de erro ao cliente a informar que o cliente é inexistente.

Por fim na figura 3.5 é testada também a função `clean_client` que já foi referido anteriormente qual é o seu objetivo, mas na mesma podemos ver os valores que devolve e perceber o seu significado. Como "abc" neste teste é o socket do cliente Manel a função devolve "True" quando este é passado como argumento, quando damos como argumento à função um socket que ainda não foi guardado ela devolve False.

```
miguel@miguel-PE60-6QE:~/Desktop/labi2021-ap2-g2/client-server/testes$ py.test-3
testNew.py
Test session starts (platform: linux, Python 3.8.5, pytest 4.6.9, pytest-sugar 0
.9.4)
rootdir: /home/miguel/Desktop/labi2021-ap2-g2/client-server/testes
plugins: sugar-0.9.4
collecting ...
testNew.py ✓ 100% ██████████
Results (0.02s):
  1 passed
```

Figura 3.6: Resultado do teste unitario

Vai ser testada a função `validate_response` do cliente, esta função consiste em validar a resposta recebida do servidor e caso na mensagem recebida o campo `status` for `True` a função devolve `True`, se a mensagem recebida conter no campo `status` o valor booleano `False`, a função deve retornar uma mensagem de erro com o erro que recebe do servidor, na figura 3.7 podemos ver o código do teste que tem todos os casos passíveis de erro.

```

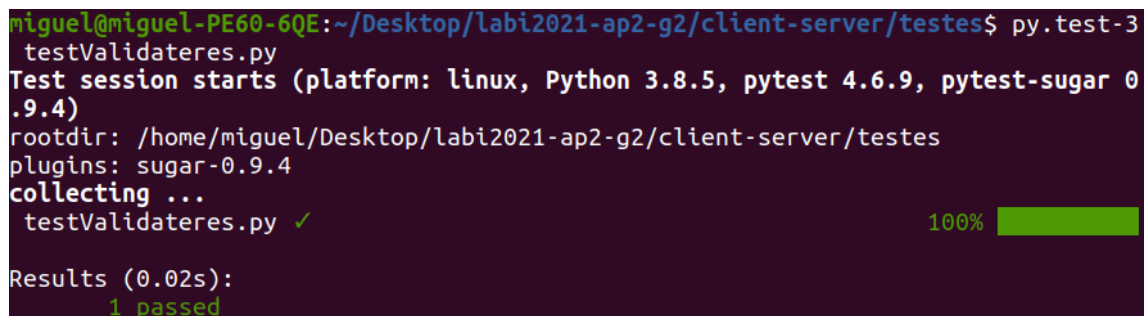
import pytest

from client import validate_response

def test():
    assert validate_response("abc", { "op": "STOP", "status":False, "error": "Cliente
    inexistente" }) == "— Erro: " + "Cliente inexistente" + " —"
    assert validate_response("abcs", { "op": "STOP", "status":False, "error": "Excedeu o
    numero maximo de tentativas" }) == "— Erro: " + "Excedeu o numero maximo de
    tentativas" + " —"
    assert validate_response("avdsefg", { "op": "STOP", "status":False, "error": "Numero
    de jogadas inconsistente" }) == "— Erro: " + "Numero de jogadas inconsistente
    " + " —"
    assert validate_response("ghjkl", { "op": "Sair", "status": False, "error": "Operação
    inexistente (operações possivesis: \ "START\ ", \ "GUESS\ ", \ "QUIT\ ", \ "STOP\ ") })
    == "— Erro: " + "Operação inexistente (operações possivesis: \ "START\ ", \ "
    GUESS\ ", \ "QUIT\ ", \ "STOP\ ") " + " —"
    assert validate_response("1234", { "op": "GUESS", "status":False, "error": "Tem de
    inserir um valor do tipo inteiro positivo" }) == "— Erro: " + "Tem de
    inserir um valor do tipo inteiro positivo" + " —"
    assert validate_response("aghsj", { "op": "GUESS", "status":False, "error": "Valor
    fora dos limites (0 <= number <= 100)" }) == "— Erro: " + "Valor fora dos
    limites (0 <= number <= 100)" + " —"
    assert validate_response("dscfew", { "op": "START", "status":True, "max_attempts":
    22 }) == True
    assert validate_response("sgdhensm", { "op": "QUIT", "status":True }) == True
    assert validate_response("fdfwd", { "op": "GUESS", "status":True, "result": 20 }) ==
    True

```

Figura 3.7: código do programa de teste



```

miguel@miguel-PE60-6QE:~/Desktop/labi2021-ap2-g2/client-server/testes$ py.test-3
testValidateres.py
Test session starts (platform: linux, Python 3.8.5, pytest 4.6.9, pytest-sugar 0
.9.4)
rootdir: /home/miguel/Desktop/labi2021-ap2-g2/client-server/testes
plugins: sugar-0.9.4
collecting ...
testValidateres.py ✓ 100%
Results (0.02s):
1 passed

```

Figura 3.8: Resultado do teste unitario

No cliente a função `quit_action` tem 2 argumentos, o primeiro o `client_sock` e o segundo a resposta do servidor, o segundo argumento não tem influência o primeiro tem para uma das operações. A função tem 3 operações a fazer, que são:

1. Fechar o socket
2. Dar return à mensagem "Desistiu com sucesso"
3. Terminar o programa do client

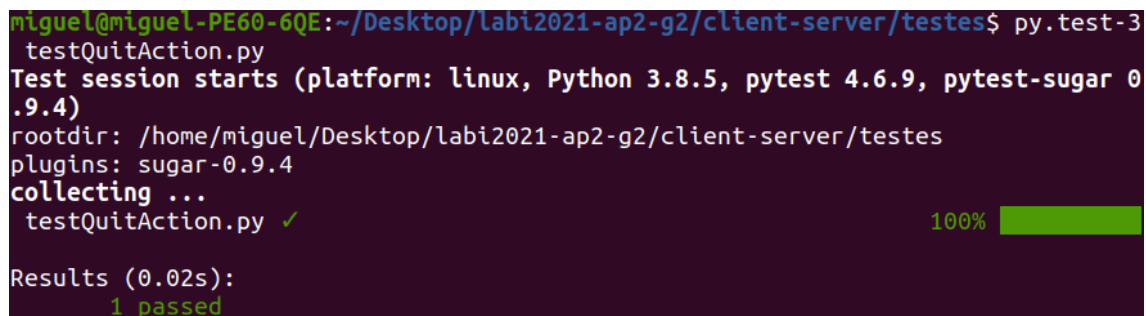
```
import pytest

from client import quit_action

def test():
    assert quit_action("abcd", 3) == "— Desistiu com sucesso —"
    assert quit_action("dfghb", 7) == "— Desistiu com sucesso —"
```

Figura 3.9: código do programa de teste

Na figura 3.9 está o código que faz a verificação do return da função `quit_action`.



```
miguel@miguel-PE60-6QE:~/Desktop/lab2021-ap2-g2/client-server/testes$ py.test-3
testQuitAction.py
Test session starts (platform: linux, Python 3.8.5, pytest 4.6.9, pytest-sugar 0
.9.4)
rootdir: /home/miguel/Desktop/lab2021-ap2-g2/client-server/testes
plugins: sugar-0.9.4
collecting ...
testQuitAction.py ✓ 100% ██████████
Results (0.02s):
 1 passed
```

Figura 3.10: Resultado do teste unitario

3.3 Simulação de uma interação cliente-servidor

Em vista simular uma possível interação entre cliente e servidor, usou-se a funcionalidade `ipdb`, assim podendo ter mais acesso em tempo real às variáveis do programa. A partir do par de imagens a baixo colocadas, pode-se verificar:

1. É inicialmente perguntado se o cliente quer usar encriptação, essa resposta pode ser dada tanto em maiúscula como em minúscula, seguida ou precedida por espaços, como qualquer resposta durante a interação.
2. Depois de enviado o pedido de **START**, que irá registar cliente na lista de clientes do servidor, uma resposta é recebida onde se verifica o sucesso da operação.
3. O jogo começa e o número máximo de jogadas é apresentado ao cliente.
4. O cliente depois de introduzir um número, esse é armazenado na variável **number**. Depois do pedido de **GUESS** pela parte do programa do cliente, o servidor responde com um dicionário de sucesso e o resultado da comparação entre o número dado pelo cliente e o número secreto, esse mesmo é imprimido no ecrã do jogador.
5. O número de tentativas efetuadas é atualizado, neste caso fez-se o print do registo do lado do cliente que é crucial estar correto, pois tem de ser igual à contagem feita pelo servidor.
6. Finalmente, quando se acerta no número secreto, neste caso o 24, a operação **STOP** é requerida ao servidor, que envia, neste caso, uma resposta afirmativa, assim, desencadeado a apresentação do resultado do jogo, o fecho do socket cliente e terminação do programa.

```

3
----> 4 import os
5 import sys

ipdb> break 117
Breakpoint 1 at /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/1abi/ap2/1abi2021-ap2-g2/client-server/client.py:117
ipdb> continue
--> Deseja usar incryptação? [Y/N] INFO para mais informações: n
> /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/1abi/ap2/1abi2021-ap2-g2/client-server/client.py(117)run_client()
116         response = sendrecv_dict(client_sock, {"op": "START", "client_id": sys.argv[1], "cipher": None})
1-> 117         print("*** Tem no maximo " + str(response["max_attempts"]) + " tentativas ***")
118         attempts = 0

ipdb> print(response)
{'op': 'START', 'status': True, 'max_attempts': 24}
ipdb> break 141
Breakpoint 2 at /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/1abi/ap2/1abi2021-ap2-g2/client-server/client.py:141
ipdb> continue
*** Tem no maximo 24 tentativas ***
--> Faça uma tentativa ou, se desejar, desista(QUIT): 50
> /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/1abi/ap2/1abi2021-ap2-g2/client-server/client.py(141)run_client()
140         attempts += 1
2-> 141         print ("*** %s ***" % response["result"])
142         if response["result"] == "equals":

ipdb> print(number)
50
ipdb> print(response)
{'op': 'GUESS', 'status': True, 'result': 'larger'}
ipdb> print(attempts)
1

```

```

--> Faça uma tentativa ou, se desejar, desista(QUIT): 24
> /mnt/c/Users/usuario/Desktop/diogo/uni/repositorios/1abi/ap2/1abi2021-ap2-g2/client-server/client.py(141)run_client()
    140         attempts += 1
2-> 141         print ("*** %s ***" % response["result"])
    142         if response["result"] == "equals":

ipdb> continue
*** equals ***
> /mnt/c/Users/usuario/Desktop/diogo/uni/repositorios/1abi/ap2/1abi2021-ap2-g2/client-server/client.py(145)run_client()
    144         validate_response(client_sock, response)
3-> 145         stop_action(client_sock, attempts)
    146         break

ipdb> print(response)
{'op': 'STOP', 'status': True, 'guess': 24}
ipdb> continue
--- Parabéns!!! Terminou o jogo em 7 tentativas ---
The program exited via sys.exit(). Exit status: 0

```

Figura 3.11: Resultado da simulação

Esta simulação possibilita, tanto a visualização de um possível uso dos programas criados, como o funcionamento dos mesmos.

Aproveito para mostrar uma função adicional criada chamada **INFO**. Quando o cliente quiser mais informação sobre o funcionamento do jogo, basta digitar INFO (ou info) e será-lhe apresentado um pequeno texto(ver figura 3.12) que poderá esclarecer qualquer dúvida existente.

```

--> Deseja usar incryptação? [Y/N] INFO para mais informações: INFO

----- INFO -----

Objetivo do jogo:

Irá lhe ser atribuído um numero maximo de jogadas,
o seu objetivo é, dentro desse limite, encontrar o numero secreto
que está compreendido entre 0 e 100

Como jogar:

1- Escolha se quer que as suas mensagens sejam encriptadas ao serem enviadas para o servidor
2- Tente adivinhar o numero, digitando o valor
3- Se em algum momento pretender desistir, digite QUIT
4- Por fim, quando acertar, será lhe apresentado uma mensagem final
5- Se ultrapassar o maximo de jogadas, quando descobrir o numero será lhe apresentado uma mensagem de erro

----- INFO -----

```

Figura 3.12: Resultado da operação **INFO**

3.4 Segurança - uso de incryptação

De forma a tornar a comunicação entre o cliente e o servidor mais segura, quando o programa cliente é iniciado é perguntado se quer usar incryptação, ou não. Em caso que o cliente o deseje, todos os numeros trocados entre o programa cliente e servidor são todos protegidos a partir de incryptação. basicamente antes de se enviar o dicionario para o respetivo destinatario, usa-se a função `encrypt_intvalue` para incriptar os valores pretendidos, para isso é criada

uma cifra de 16 bytes aleatoria, que entra como argumento na função, para além do valor a incryptar. No lado do destinatário a função **decrypt_intvalue** descripta o valor para, assim, poder ser usado da forma típica.

Para testar este sistema de incryptação / desincryptação usou-se a funcionalidade **ipdb** que nos permitiu visualizar o valor de variáveis durante o funcionamento do programa. Estes testes foram feitos no programa cliente, sendo possível verificar a correção do servidor a partir das respostas recebidas.

```

3
----> 4 import os
5 import sys

ipdb> break 83
Breakpoint 1 at /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/labi/ap2/labi2021-ap2-g2/client-server/client.py:83
ipdb> continue
--> Deseja usar incryptação? [Y/N] INFO para mais informações: Y
> /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/labi/ap2/labi2021-ap2-g2/client-server/client.py(83)run_client()
82         response = sendrecv_dict(client_sock, {"op": "START", "client_id": sys.argv[1], "cipher": cipherkey_tosend
1--> 83         print("*** Tem no maximo ", str decrypt_intvalue(response["max_attempts"], cipher), " tentativas ***")
84         attempts = 0

ipdb> print(cipherkey_tosend)
0JXhyvAkccfZs4gvjbqmFA==
ipdb> print(response)
{'op': 'START', 'status': True, 'max_attempts': 'YQZ/mZy1QPGJGN67Q0+wTQ=='}
ipdb> break 103
Breakpoint 2 at /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/labi/ap2/labi2021-ap2-g2/client-server/client.py:103
ipdb> continue
*** Tem no maximo 28 tentativas ***
--> Faça uma tentativa ou, se desejar, desista(QUIT): 25
> /mnt/c/Users/utilizador/Desktop/diogo/uni/repositorios/labi/ap2/labi2021-ap2-g2/client-server/client.py(103)run_client()
102         number = encrypt_intvalue(cipher, number)
2--> 103         response = sendrecv_dict(client_sock, {"op": "GUESS", "number": number})
104

ipdb> print(number)
spP9ggjomCLkpeFb/eD4EA==

```

Figura 3.13: Resultado do teste ipdb da segurança

Como é possível visualizar na imagem acima, uma cifra é criada, ver resultado de **print(cipherkey_tosend)**, e as respostas do servidor também veem encriptadas, ver **print(response)** onde aparece o número máximo de tentativas encriptado, logo a seguir a mesma é descriptada, ver a mensagem onde aparece o 28 como número máximo de tentativas. Para reforçar a ideia, ainda se fez o **print(number)** que seria, em decimal, 25, mas aparece , depois de encriptado, como “spP9ggjomCLkpeFb/eD4EA==”.

Depois deste pequeno teste, conclui-se que o sistema de encriptação / desencriptação está a funcionar como esperado.

Capítulo 4

Análise geral

Resumindo, o que foi possível concluir depois dos testes e simulações realizadas? Primeiramente, asseguramos que a iniciação dos programas criados são o mais robustos possível, tratando os erros gerados de forma a se tornarem em mensagens que possam elucidar o cliente. Depois de todos os programas a correr, era crucial que as funções implementadas servissem para o que forma desenhadas, na Seção 3.2 asseguramos que as funções essenciais funcionam como esperado, tanto a nível de retornos como de deteção de erros. De seguida na autorefsec.simulação mostramos aquilo que poderia ser uma interação cliente / servidor, usando a ferramenta ipdb do python fomos um pouco mais além e mostramos que realmente as variáveis e dicionários, que vão flutuando durante o percorrer dos programas, eram atualizadas. Por fim, testou-se a parte da segurança, o sistema de encriptação e de desencriptação tinham de estar a funcionar de forma perfeita, assim fazendo com que a comunicação não fosse perdida ou modificada em qualquer momento, assim, viu-se que tal está assegurado.

Capítulo 5

Conclusões

Agora, estamos na posição de afirmar que os programas funcionam como esperado, a robustez têm bastante abrangência e uma comunicação mais segura entre cliente / servidor também é possível. Os objetivos delineados para este projeto foram alcançados com sucesso.

Contribuições dos autores

As funções essenciais foram maioritariamente feitas por DM, a implementação da segurança e das funções de escrita do report csv foram realizadas por MA. O relatório foi escrito sem grande distensão.

Contribuições: DM- 50%, MG- 50%.

Acrónimos

LABI Laboratórios de Informática

UC unidade curricular

CSV Comma-separated values

Bibliografia

- [1] U. de Aveiro, «Trabalho de aprofundamento 2», mai. de 2021. URL: https://elearning.ua.pt/pluginfile.php/430960/mod_resource/content/13/ap2-2021.pdf.