

Adivinha o Número Secreto

Universidade de Aveiro

Mauro Marques Canhão Filho, Patricia Rafaela
da Rocha Cardoso



Adivinha o Número Secreto

Departamento de Eletrônica, Telecomunicações e
Informática (DETI)

Universidade de Aveiro

Mauro Marques Canhão Filho, Patricia Rafaela da Rocha Cardoso
(103411) mauro.filho@ua.pt, (103243) patriciarcardoso@ua.pt

30/05/2020

Resumo

Este relatório tem como objetivo descrever a implementação e a interação entre um servidor e um ou mais clientes. Para isso, será detalhadamente apresentado o funcionamento/criação de um jogo. O jogo consiste em o cliente adivinhar um número inteiro aleatório entre 0 e 100, o número secreto, gerado aleatoriamente pelo servidor. Serão detalhadamente descritos o programa cliente e o programa servidor e a criptografia a eles associados bem como o funcionamento criação e desenvolvimento de testes funcionais e unitários. Com base em imagens de todas as funções em ambos os programas e imagens da interação no terminal entre um ou mais clientes e o servidor é possível perceber o funcionamento correto da aplicação.

Índice

1	Introdução	1
2	Metodologia	2
2.1	Servidor	2
2.1.1	Armazenamento dos resultados num ficheiro csv	2
2.1.2	Funcionamento geral do jogo	4
2.1.3	Segurança	9
2.1.4	Main	11
2.2	Cliente	11
2.2.1	Funcionamento geral do jogo	11
2.2.2	Segurança	18
2.2.3	Main	19
2.3	Testes unitários e funcionais	21
2.3.1	Testes Funcionais	21
2.3.2	Teste unitário	23
3	Resultados/Análise	25
4	Conclusões	29

Lista de Figuras

2.1	Função que cria um ficheiro report.csv quando o servidor é inicializado.	2
2.2	Dicionário constituído pelos dados dos jogadores e array responsável pela inicialização do header no ficheiro report.csv.	3
2.3	Função que atualiza o ficheiro report.csv quando um jogo é terminado.	3
2.4	Função que cria um novo cliente no jogo.	4
2.5	Função que retorna o porto ao qual o cliente está conectado. . . .	4
2.6	Função chamada sempre que o servidor recebe uma nova mensagem do cliente.	5
2.7	Suporte da jogada de um cliente - Operação GUESS.	6
2.8	Função que devolve o número secreto.	6
2.9	Função chamada quando o cliente pretende desistir do jogo. . . .	7
2.10	Função responsável por encerrar o jogo.	8
2.11	Função chamada sempre que é necessário apagar um jogador da lista de jogadores ativos.	9
2.12	Função para encriptar valores a enviar em formato JSON com codificação base64.	9
2.13	Função para desencriptar valores recebidos em formato json com codificação base64.	10
2.14	Função que permite o funcionamento correto de todo o servidor. .	11
2.15	Função que valida a resposta do servidor recebida pelo cliente. .	12
2.16	Variáveis da função run_client	12
2.17	Função run_client	13
2.18	Caso em que é inserido o comando START.	13
2.19	Caso em que é inserido o comando QUIT.	14
2.20	Função quit_action	14
2.21	Caso em que é inserido o comando GUESS.	15
2.22	Caso em que é inserido o comando GUESS.	15
2.23	Caso em que é inserido o comando GUESS.	16
2.24	Caso em que é inserido o comando GUESS.	17
2.25	Caso em que é inserido o comando GUESS.	17
2.26	Caso em que é inserido o comando STOP.	17
2.27	Função para encriptar valores a enviar em formato JSON com codificação base64.	18

2.28	Chave de cifragem gerada e codificada no modelo Base64 para ser enviada ao servidor.	18
2.29	Função para descriptar valores recebidos em formato JSON com codificação base64.	19
2.30	Chave de cifragem gerada e codificada no modelo Base64 para ser enviada ao servidor.	19
2.31	Função main	20
2.32	Função main	21
2.33	Função main	21
3.1	Terminal do cliente.	25
3.2	Terminal do servidor.	26
3.3	Terminal do cliente.	26
3.4	Terminal do servidor.	27
3.5	Ação da aplicação quando existem dois clientes com o mesmo id.	27
3.6	Ação da aplicação quando existem dois clientes diferentes a jogar simultaneamente.	27
3.7	Comportamento da aplicação perante introdução de comandos incorrectos.	28
3.8	Ficheiro report.csv	28

Capítulo 1

Introdução

O objetivo deste trabalho é explicar, enumerar e descrever o desenvolvimento e funcionamento de um servidor que suporte a geração de um número inteiro aleatório (entre 0 e 100), o 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 nunca deverá aceitar dois clientes com a mesma identificação a jogar simultaneamente e deverá criar e atualizar um ficheiro designado por report.csv onde vai escrevendo os resultados dos diversos clientes quando estes terminam o jogo. O cliente pode desistir em qualquer altura e o jogo acaba quando ele adivinha o número secreto ou quando esgota o número máximo de tentativas que dispunha para jogar. Caso o cliente exceda o número de jogadas de que dispunha o jogo será considerado sem sucesso mesmo que ele tenha adivinhado o número. Quando o jogo acaba corretamente o cliente deve escrever no monitor uma mensagem a indicar se adivinhou ou não o número secreto e quantas jogadas efectuou. Por sua vez o servidor acrescenta ao ficheiro a informação relativa ao jogo: cliente; número secreto; número máximo de jogadas; número de jogadas efectuadas; e o resultado obtido pelo cliente (desistência ou sucesso ou insucesso). Também será abordado e aprofundado o conceito de criptografia e a realização e funcionamento de testes funcionais e unitários bem como a explicação detalhada do desenvolvimento da aplicação.

Capítulo 2

Metodologia

Neste capítulo será detalhadamente descrito o algoritmo e o funcionamento do programa servidor e do programa cliente bem como a implementação dos testes funcionais e unitários.

2.1 Servidor

O programa servidor consiste em gerar aleatoriamente um número entre 0 e 100 e um número máximo de tentativas entre 10 e 30 para o adivinhar. O programa servidor é constituído por um dicionário e as seguintes funções: **find_client_id**, **encrypt_intvalue**, **decrypt_intvalue**, **new_msg**, **numberToCompare**, **new_client**, **clean_client**, **quit_client**, **create_file**, **update_file**, **guess_client**, **stop_client** e **main**.

2.1.1 Armazenamento dos resultados num ficheiro csv

```
...  
  
def create_file():  
    if path.exists('report.csv') == False:  
        with open('report.csv', 'w') as fileCSV:  
            writer = csv.DictWriter(fileCSV, fieldnames=header)  
            writer.writeheader()  
        return None  
  
...
```

Figura 2.1: Função que cria um ficheiro report.csv quando o servidor é inicializado.

No momento em que o servidor é inicializado é chamada a função "create_file" para que seja criado um novo ficheiro report.csv caso ainda não exista no diretório em que o server.py se encontra. Assim, o servidor não reinicia o ficheiro sempre

que for inicializado. Depois, escreve o cabeçalho no ficheiro com base no array "header". O array "header" é utilizado para atualizar o cabeçalho do ficheiro report.csv que será gerado pelo servidor.

```
...
gamers = {'name':[], 'sock_id':[], 'segredo':[], 'max':[], 'jogadas':[], 'resultado':[], 'cipherkey':[]}
header = ['name', 'sock_id', 'segredo', 'max', 'jogadas', 'resultado']
...
```

Figura 2.2: Dicionário constituído pelos dados dos jogadores e array responsável pela inicialização do header no ficheiro report.csv.

Por outro lado, o dicionário "gamers" armazena os dados dos jogadores que estão atualmente com um jogo iniciado. A informação armazenada é baseada na ordem pela qual os clientes se conectam ao servidor. Essa informação é filtrada e distribuída por arrays que contêm diferentes campos de identificação. Por exemplo, se dois jogadores, Mauro e Patrícia estiverem a jogar simultaneamente e se o Mauro se conectou primeiro ao servidor, o seu ID pode ser consultado através de: gamers['sock_id'][0], enquanto o ID da Patrícia pode ser acedido da seguinte forma: gamers['sock_id'][1].

```
...
def update_file(client_id, result):
    with open('report.csv', 'a') as fileCSV:
        writer = csv.DictWriter(fileCSV, fieldnames=header)
        for i in range(0, len(gamers['sock_id'])):
            if client_id == gamers['sock_id'][i]:
                di = { 'name': gamers['name'][i], 'sock_id': gamers['sock_id'][i],
                      'segredo': gamers['segredo'][i], 'max': gamers['max'][i],
                      'jogadas': gamers['jogadas'][i], 'resultado': result}
                writer.writerow(di)
    return None
...
```

Figura 2.3: Função que atualiza o ficheiro report.csv quando um jogo é terminado.

Quando um jogo termina com sucesso, sem sucesso ou em caso de desistência é chamada a função "update_file" que atualiza o ficheiro report.csv com os dados do jogador. Para isso, abre o ficheiro no modo "a" (append) para adicionar dados sem escrever sobre aqueles que já lá estavam. Assim, procura pelo index "i" tal que o sock_id é igual ao client_id passado como parâmetro da função. Por fim, escreve todos os itens na posição "i" dos arrays do dicionário "gamers" no ficheiro report.csv.

2.1.2 Funcionamento geral do jogo

```
...
def new_client (client_sock, request):
    name = request['client_id']
    sock_id = find_client_id(client_sock)
    if name in gamers['name']:
        response = {'op': "START", 'status': False, 'error': "Cliente existente"}
        send_dict(client_sock, response)
    else:
        gamers['name'].append(name)
        gamers['sock_id'].append(sock_id)
        n = random.randint(10, 30)
        secret = random.randint(0, 100)
        gamers['segredo'].append(secret)
        gamers['max'].append(n)
        gamers['jogadas'].append(0)
        gamers['cipherkey'].append(base64.b64decode(request['cipherkey']))
        print(gamers)
        response = {'op': "START", 'status': True, 'max_attempts': encrypt_intvalue(sock_id,n)}
        send_dict(client_sock, response)
    return None
...
```

Figura 2.4: Função que cria um novo cliente no jogo.

O jogo é iniciado quando o servidor recebe uma mensagem do cliente com o comando "START" tornando-se num jogador ativo e provocando as seguintes ações no servidor:

1. Armazenamento na variável "name" do "client_id" passado para o servidor aquando da inserção pelo utilizador na linha de comandos ao executar o cliente;
2. Identificação do ID(porto ao qual está conectado) do cliente a partir do socket recorrendo à função "find_client_id";

```
...
def find_client_id (client_sock):
    peerName = client_sock.getpeername()
    return peerName[1]
...
```

Figura 2.5: Função que retorna o porto ao qual o cliente está conectado.

A partir de cada socket de cliente, é possível extrair algumas informações únicas para o identificar. Neste caso, a função `.getpeername()` devolve um tuplo que contém o endereço do host e o porto ao qual o cliente está conectado. O porto, por sua vez, é devolvido pela função `find_client_id()`.

3. Envio de uma resposta do servidor para o cliente com status: True; e com o valor encriptado de jogadas máximas que o cliente pode fazer.

Se "name"("client_id"enviado pelo pedido do cliente) já se encontrar no dicionário "gamers", o servidor irá relatar ao cliente uma mensagem de status: False; e uma mensagem de erro indicando a já utilização desse nome. Caso contrário, a função adiciona todos os dados necessários do cliente aos arrays do dicionário. É depois, iniciado um jogo.

```
...
def new_msg (client_sock):
    request = recv_dict(client_sock)
    print(request)
    if request['op'] == "START":
        new_client(client_sock, request)
    if request['op'] == "QUIT":
        quit_client(client_sock)
    if request['op'] == "STOP":
        stop_client(client_sock, request)
    if request['op'] == "GUESS":
        guess_client(client_sock, request)
    return None
...
```

Figura 2.6: Função chamada sempre que o servidor recebe uma nova mensagem do cliente.

Seguidamente o jogador terá que introduzir uma das seguintes operações na linha de comandos: GUESS, STOP ou QUIT. A tarefa desta função é identificar qual a operação requisitada pelo cliente e encaminhá-la para a função que irá processar e responder ao pedido. Caso seja feito um pedido de uma operação fora do alcance da aplicação não ocorre qualquer comportamento por parte do servidor.

```

...
def guess_client (client_sock, request):
    if find_client_id(client_sock) in gamers['sock_id']:
        segredo = numberToCompare(client_sock)
        jogado = decrypt_intvalue(find_client_id(client_sock),request['number'])

        if jogado == segredo:
            response = {'op': "GUESS", 'status': True, 'result':"equals"}
            send_dict(client_sock, response)
        if jogado > segredo:
            response = {'op': "GUESS", 'status': True, 'result':"larger"}
            send_dict(client_sock, response)
        if jogado < segredo:
            response = {'op': "GUESS", 'status': True, 'result':"smaller"}
            send_dict(client_sock, response)
        for i in range(0, len(gamers['sock_id'])):
            if find_client_id(client_sock) == gamers['sock_id'][i]:
                gamers['jogadas'][i] = gamers['jogadas'][i] + 1
    else:
        response = {'op': "GUESS", 'status': False, 'error': "Client inexistente"}
        send_dict(client_sock, response)

    return None
...

```

Figura 2.7: Suporte da jogada de um cliente - Operação GUESS.

O utilizador deve introduzir o comando GUESS antes de inserir o seu palpite. No entanto, é essencial averiguar se o cliente que está a jogar tem realmente uma sessão iniciada no jogo.

Se o jogador estiver presente no dicionário "gamers" prosseguimos com o GUESS. Caso contrário, o servidor envia uma mensagem ao cliente com o status: False; e uma mensagem de erro a indicar que este não se encontra na lista de jogadores ativos.

Consideremos agora o caso em que o cliente tem um jogo iniciado. Primeiro, procuramos o valor do número secreto deste cliente através da função "numberToCompare()", que será armazenado na variável segredo.

```

...
def numberToCompare(client_sock):
    id = find_client_id(client_sock)
    for i in range(0, len(gamers['sock_id'])):
        if gamers['sock_id'][i] == id:
            return gamers['segredo'][i]
    ...

```

Figura 2.8: Função que devolve o número secreto.

Depois, descriptografamos o número inserido pelo jogador (que é passado na mensagem enviada do cliente ao servidor e que depois é encaminhada para a função pelo parâmetro "request") que é armazenado na variável "jogado".

- Se o número for igual ao número secreto, o servidor envia uma mensagem

ao cliente com status: True e result: "equals", a indicar que o jogador acertou no número;

- Se o número for maior que o segredo, o servidor envia uma mensagem ao cliente com status: True e result: "larger" a indicar que o jogador introduziu um número superior ao número secreto;
- Se o número for menor que o segredo, o servidor envia uma mensagem ao cliente com status: True e result: "smaller", a indicar que o jogador introduziu um número mais pequeno que o número secreto;

Por fim, atualiza no dicionário "gamers" o número de jogadas efetuadas.

```
...
def quit_client (client_sock):
    if find_client_id(client_sock) in gamers['sock_id']:
        response = {'op': "QUIT", 'status': True}
        send_dict(client_sock, response)
        update_file(find_client_id(client_sock), 'DESISTENCIA')
        clean_client(client_sock)
    else:
        response = {'op': "QUIT", 'status': False, 'error': "cliente inexistente"}
        send_dict(client_sock, response)
    print("CURRENT GAMERS: "+str(gamers))
    return None
...
```

Figura 2.9: Função chamada quando o cliente pretende desistir do jogo.

Caso o jogador queira desistir do jogo, o servidor recebe uma mensagem com o comando QUIT. Isto induz a função "quit_client" a conferir se o cliente que pretende desistir encontra-se realmente em jogo. Para isto, verifica se o ID do socket está presente no dicionário "gamers".

Em caso afirmativo, o servidor envia uma mensagem ao cliente com status: True; e atualiza o ficheiro report.csv (recorrendo à função update_file()) com o resultado "DESISTENCIA". Este resultado indica que a partida foi terminada antes de o jogador adivinhar o número secreto ou antes de atingir o limite de jogadas. Por fim, remove o cliente da lista de jogadores ativos recorrendo à função clean_client.

Caso contrário, envia uma mensagem ao cliente com status: False; e uma mensagem de erro que explicita o facto de o cliente não ter sido encontrado entre os jogadores ativos.

```

...
def stop_client (client_sock, request):
    if find_client_id(client_sock) in gamers['sock_id']:
        response = {'op': "STOP", 'status': True}
        send_dict(client_sock, response)
        for i in range(0, len(gamers['sock_id'])):
            if find_client_id(client_sock) == gamers['sock_id'][i]:
                gamers['jogadas'][i] = decrypt_intvalue(gamers['sock_id'][i], request['attempts'])
                if gamers['segredo'][i] == decrypt_intvalue(gamers['sock_id'][i], request['number']):
                    update_file(find_client_id(client_sock), "SUCCESS")
                else:
                    update_file(find_client_id(client_sock), "FAILURE")
            clean_client(client_sock)
    else:
        response = {'op': "STOP", 'status': False, 'error': "cliente inexistente"}
        send_dict(client_sock, response)
    print("CURRENT GAMERS: " + str(gamers))
    return None
...

```

Figura 2.10: Função responsável por encerrar o jogo.

Quando um jogo é terminado ou porque o jogador acertou no número secreto ou porque efetuou mais jogadas dos que as que possuía é executada a função "stop_client".

Para que um jogo seja encerrado, o cliente precisa estar na lista de jogadores ativos, ou seja, no dicionário "gamers". Se o cliente não se encontrar ativo no jogo, a função envia-lhe uma mensagem com status: False e uma mensagem de erro a indicar que o cliente não se encontra na lista de jogadores ativos. Caso o cliente esteja ativo no jogo, o servidor envia-lhe uma mensagem com status: True, a indicar que a finalização do jogo foi processada.

O processamento da finalização do jogo dá-se da seguinte forma:

1. O servidor atualiza no dicionário "gamers" o número de jogadas efetuadas pelo jogador. Para isso, deve descriptografar o número inteiro enviado pelo cliente com auxílio da função "decrypt_intvalue()";
2. O servidor verifica se o último número jogado pelo utilizador (que também deve ser descriptografado) coincide com o número secreto. Em caso afirmativo, atualiza o ficheiro report.csv com os dados do cliente e o resultado final "SUCCESS". Caso contrário, atualiza o ficheiro report.csv com os dados do cliente e o resultado final "FAILURE";
3. Elimina o cliente da lista de jogadores ativos através da função "clean_client()".

```

...
def clean_client (client_sock):
    id = find_client_id(client_sock)
    print("numero de gamers: " + str(len(gamers['sock_id'])))
    for i in range(0, len(gamers['sock_id'])):
        print("index: "+str(i))
        if gamers['sock_id'][i] == id:
            gamers['segredo'].pop(i)
            gamers['sock_id'].pop(i)
            gamers['name'].pop(i)
            gamers['max'].pop(i)
            gamers['jogadas'].pop(i)
            gamers['cipherkey'].pop(i)
        return True
    return False
...

```

Figura 2.11: Função chamada sempre que é necessário apagar um jogador da lista de jogadores ativos.

Esta função é executada sempre que for necessário excluir um cliente do dicionário "gamers". Isto ocorre quando o cliente se desconecta do servidor, quando termina o jogo ou quando desiste. A função procura pelo cliente no dicionário "gamers" e caso o encontre, exclui todos os dados a ele associados através do seu respectivo índice.

2.1.3 Segurança

Encriptação

```

...
def encrypt_intvalue (client_id, data):
    for i in range(0, len(gamers['sock_id'])):
        if gamers['sock_id'][i] == client_id:
            cipherkey = gamers['cipherkey'][i]

    cipher = AES.new(cipherkey, AES.MODE_ECB)
    data2 = cipher.encrypt(bytes("%16d" % (data), 'utf8'))
    data_tosend = str(base64.b64encode(data2), 'utf8')
    return data_tosend
...

```

Figura 2.12: Função para encriptar valores a enviar em formato JSON com codificação base64.

Cada número inteiro comunicado entre o servidor e o cliente é encriptado por blocos usando a função AES-128 no modo ECB. A encriptação é realizada do seguinte modo:

1. Identificação da chave de cifragem relativa ao cliente atual comparando o ID passado como argumento da função e os IDs presentes no dicionário "gamers";

2. Conversão do inteiro numa string binária de 128 bits;
3. Codificação da string no formato Base64 com o intuito dos criptogramas serem suportados pelo JSON;
4. Devolução pela função do valor codificado e encriptado para que possa ser enviado.

Descriptação

```
...
def decrypt_intvalue (client_id, data):
    for i in range(0, len(gamers['sock_id'])):
        if gamers['sock_id'][i] == client_id:
            cipherkey = gamers['cipherkey'][i]

    cipher = AES.new(cipherkey, AES.MODE_ECB)
    data1 = base64.b64decode(data)
    data2 = cipher.decrypt(data1)
    print(data2)
    data3 = int(str(data2, 'utf8'))
    return data3
...
```

Figura 2.13: Função para descriptar valores recebidos em formato json com codificação base64.

Cada número inteiro comunicado entre o servidor e o cliente é descriptado por blocos usando a função AES-128 em modo ECB. A descriptação ocorre do seguinte modo:

1. Identificação da chave de cifragem relativa ao cliente atual comparando o ID passado como argumento da função e os IDs presentes no dicionário "gamers";
2. Descodificação dos dados passados à função como argumento no formato Base64 e descriptação do seu conteúdo;
3. Codificação para um valor inteiro;
4. Devolução do valor inteiro pela função.

2.1.4 Main

```
269 ...
270 def main():
271     if len(sys.argv) != 2:
272         print("Deve passar o porto como argumento para o servidor")
273         sys.exit(1)
274     try:
275         int(sys.argv[1])
276     except ValueError:
277         print("Porto deve ser um numero inteiro")
278         sys.exit(2)
279     if int(sys.argv[1]) < 0:
280         print("Porto deve ser um numero inteiro positivo")
281         sys.exit(2)
282
283     port = int(sys.argv[1])
284 ...
```

Figura 2.14: Função que permite o funcionamento correto de todo o servidor.

Esta função permite:

- Nas linhas 271-273 verificar se o servidor é iniciado com um argumento(porto). Caso não seja, o programa encerra com uma mensagem de erro;
- Nas linhas 274-281 verificar se o porto é um número inteiro. Se não for, o programa é encerrado com uma mensagem de erro;
- Na linha 283 é atribuído o valor do porto à variável.

2.2 Cliente

O programa cliente consiste em permitir que um utilizador jogue o jogo "Advinhe o número secreto" ao introduzir as operações START, GUESS, QUIT e o seu palpite na linha de comandos.

O programa cliente é constituído por uma chave de cifragem e pelas seguintes funções: **encrypt_intvalue**, **decrypt_intvalue**, **validate_response**, **quit_action**, **run_client** e **main**.

2.2.1 Funcionamento geral do jogo

Para que a aplicação funcione corretamente é essencial que exista uma função no programa cliente que valide a resposta enviada pelo servidor. Para isso é executada a função "validate_response".

```

49  ...
50  def validate_response_(client_sock, response):
51      try:
52          op = response['op']
53          status = response['status']
54          return True
55      except:
56          return False
57      return None
58  ...

```

Figura 2.15: Função que valida a resposta do servidor recebida pelo cliente.

A resposta do servidor é válida quando é do tipo "'op' : xxxx, 'status' : xxxx". Assim, usa-se um "try-catch" que tenta aceder aos valores de 'op' e 'status'. Se o acesso for possível é devolvido "True". Caso o acesso falhe é retornado "False".

A função que gerencia o programa cliente é a função "run_client".

```

108  #...
109  def run_client_(client_sock, client_id):
110
111      emUso = True
112      jogadas = 0
113      jogMax = None
114      auto=False
115      nextCom = ""
116      lastAttempt = None
117  #...

```

Figura 2.16: Variáveis da função **run_client**

As variáveis da função "run_client" desempenham as seguintes funcionalidades:

- emUso: Enquanto estiver True, o programa irá estar à espera de novos comandos. A função STOP pode encerrar o ciclo ao mudar esta variável para False;
- jogadas: Número de jogadas efetuadas pelo jogador;
- jogMax: Número máximo de jogadas que um jogador pode fazer. É uma variável vazia até que receba o número de tentativas enviadas pelo servidor após a introdução do comando START pelo cliente;
- auto: Os comandos START, GUESS e QUIT devem ser introduzidos pelo utilizador. O comando STOP é inserido automaticamente pelo sistema quando é detectado o fim de um jogo. Assim, quando esta variável for TRUE, o próximo comando a ser executado será feito automaticamente pelo sistema e quando for FALSE, o programa fica à espera de um input do utilizador;

- nextCom: Próximo comando a ser executado em modo automático;
- lastAttempt: Último número jogado pelo utilizador ao executar o comando GUESS.

```

122     ...
123     while emUso:
124
125         if auto == False:
126             inp = input("Comando: ")
127             comando = inp.upper()
128         else:
129             comando = nextCom
130             auto = False
131     ...

```

Figura 2.17: Função `run_client`

Neste ciclo while o programa está à espera de novos comandos a serem introduzidos pelo cliente (`emuso == True`). Logo, se `auto == False` o próximo comando a ser executado deverá ser introduzido pelo utilizador. Caso contrário, o comando executado será "nextCom".

```

145     ...
146     if comando == "START":
147
148         request = {'op': 'START', 'client_id': client_id, 'cipherkey': cipherkey_toSend }
149         response = sendrecv_dict(client_sock, request)
150
151         if validate_response(client_sock, response):
152             if response['status'] == False:
153                 print("Erro: " + response['error'])
154             else:
155                 max = decrypt_intvalue(cipherkey, response['max_attempts'])
156                 message = "Jogo iniciado, tens " + str(max) + " jogadas para acertar o segredo."
157                 jogMax = max
158                 jogadas = 0
159                 print(message)
160         else:
161             print("Erro: resposta do servidor não é válida")
162     ...

```

Figura 2.18: Caso em que é inserido o comando START.

Por outro lado, se for introduzido o comando START, o cliente envia uma mensagem ao servidor indicando a operação que deseja executar, o id do cliente (inserido na linha de comandos ao executar o programa), e a chave de cifra codificada. Ao receber uma resposta, esta será validada pela função "validate_response" e caso seja válida:

1. Verifica o status da resposta. Se for False, deverá imprimir uma mensagem de erro recebida do servidor. Se for True irá descriptografar o número máximo de jogadas para esta partida recorrendo à função "decrypt_intvalue";
2. Inicia a contagem de jogadas, atribuindo o valor zero à variável "jogadas";
3. Mostra uma mensagem de confirmação ao utilizador.

Se a resposta do servidor não for válida:

1. Mostra uma mensagem de erro referente ao erro identificado pelo servidor.

```

167 ...
168 if comando == "QUIT":
169     quit_action(client_sock, jogadas)
170     jogadas = 0
171     jogMax = None
172 ...

```

Figura 2.19: Caso em que é inserido o comando QUIT.

Se o comando introduzido for o QUIT, o programa processa o pedido do cliente recorrendo à função "quit_action()" e as variáveis "jogadas" e "jogMax" voltam aos seus estados iniciais.

```

69 ...
70 def quit_action(client_sock, attempts):
71     request = {'op': 'QUIT'}
72     response = sendrecv_dict(client_sock, request)
73
74     if validate_response(client_sock, response):
75         if response['status']:
76             print("Jogo terminado com sucesso.")
77         else:
78             print("Impossível terminar um jogo que não foi iniciado.")
79     else:
80         print("Erro: resposta do servidor não é válida")
81     return None
82 ...

```

Figura 2.20: Função **quit_action**.

A função **quit_action** processa a operação QUIT. Quando o utilizador introduzir o comando QUIT no terminal esta função irá enviar uma mensagem ao servidor e aguardar por uma resposta. Esta resposta é devolvida pela função "sendrecv_dict()".

Depois, a função **quit_action** armazena a resposta na variável "response". Seguidamente, recorrendo à função "validate_response" verifica se a resposta é válida. Em caso afirmativo verifica o 'status' da resposta. Se o status for TRUE o jogo foi terminado com sucesso (imprime uma mensagem de sucesso). Se o status for False,

indica que o servidor não encontrou o cliente atual na lista de jogadores ativos e imprime uma mensagem de erro. Caso a resposta do servidor não seja válida, é imprimida uma mensagem de erro correspondente.

```
179 ...
180 if comando == "GUESS":
181     numeroValido = False
182     numeroInteiro = False
183     number = ""
184
185
186 while numeroInteiro != True or numeroValido != True:
187     numeroValido = False
188     numeroInteiro = False
189
190     number_str = input("Numero para jogar: ")
191
192     try:
193         number = int(number_str)
194         numeroInteiro = True
195     except:
196         print("Jogada deve ser um numero")
197         continue
198
199     number = int(number_str)
200
201     if number >= 0 and number <= 100:
202         numeroValido = True
203     else:
204         print("Numero deve estar entre 0 e 100")
205         continue
206 ...
```

Figura 2.21: Caso em que é inserido o comando GUESS.

Se o comando introduzido for GUESS, têm que ser inicializadas duas novas variáveis para verificar se o número introduzido pelo utilizador é de facto um número inteiro e se é válido (se está entre 0 e 100). Nas linhas 186-205 é verificado se o valor introduzido é um inteiro válido e se não for, o programa pede um novo valor ao utilizador.

```
215 ...
216 request = {'op': 'GUESS', 'number': encrypt_intvalue(cipherkey, number)}
217 lastAttempt = number
218
219 if jogadas == jogMax:
220     continue
221
222 ...
```

Figura 2.22: Caso em que é inserido o comando GUESS.

Tendo já a certeza de que o palpite introduzido pelo cliente é um número inteiro válido prepara-se a mensagem que será enviada ao servidor. A mensagem contém o número jogado após ser criptografado recorrendo à função "encrypt_intvalue()".

Na linha 217 a variável "lastAttempt" recebe o número que foi jogado. Na linha 219 é verificado se o número de jogadas é igual ao número máximo de jogadas. Se sim, nada ocorre e o ciclo passa para a próxima execução.

```
235 ...
236 else:
237     response = sendrecv_dict(client_sock, request)
238
239     if validate_response(client_sock, response):
240
241         if response['status']:
242             jogadas = jogadas + 1
243
244             if response['result'] == "equals":
245                 print("Parabéns, acertaste o número!")
246                 auto = True
247                 nextCom = "STOP"
248                 continue
249
250             elif response['result'] == "smaller":
251                 print("O número secreto é maior!")
252
253             elif response['result'] == "larger":
254                 print("O número secreto é menor!")
255
256         print("Max: " + str(jogMax) + " Jogadas: " + str(jogadas))
257
258 ...
259
```

Figura 2.23: Caso em que é inserido o comando GUESS.

Caso o jogador ainda tenha jogadas disponíveis a mensagem é enviada ao servidor e a sua resposta é validada pela função "validate_response()". Nas linhas 241-242 verificamos que se a resposta for válida e se o status for True, ou seja, se o servidor não reportar erros, a resposta é processada sendo somado 1 ao número de jogadas. De seguida, é avaliado o resultado reportado pelo servidor.

Se o resultado for "equals", vemos pelas linhas 244-248, que o jogador acertou no número. E, portanto é atribuído True à variável "auto" e STOP à "nextCom" ficando implícito que o jogo acabou e que o comando STOP será executado automaticamente na próxima execução do ciclo while.

Nas linhas 250-251 é retratado que se o resultado for "smaller" é exibida uma mensagem "O número secreto é maior!". E, nas linhas 254-255 é imprimida uma mensagem "O número secreto é menor!" quando o resultado for "larger".

No fim de cada tentativa é lembrado ao jogador o número máximo de jogadas que pode fazer e o número de jogadas que já efetuou - linha 258.

```

261
262
263
264
265
266
267
...
if jogadas == jogMax:
    print("Atingiu o limite de jogadas! Comece outro jogo para tentar novamente.")
    auto = True
    nextCom = "STOP"
    continue
...

```

Figura 2.24: Caso em que é inserido o comando GUESS.

Se o número de jogadas efetuadas for igual ao número máximo de jogadas, é mostrada uma mensagem "Atingiu o limite de jogadas! Comece outro jogo para tentar novamente" e é atribuído True à variável "auto" e STOP à variável "nextCom".

```

272
273
274
275
276
277
...
else:
    print(response['error'])
else:
    print("Erro: resposta do servidor não é válida")
...

```

Figura 2.25: Caso em que é inserido o comando GUESS.

Se o status da mensagem recebida do servidor for False, é exibida uma mensagem contendo o erro reportado ao servidor - linhas 273-274. Se a resposta do servidor não for válida, é mostrada a respectiva mensagem de erro - linhas 275-276.

```

282
283
284
285
286
287
288
289
290
291
292
293
294
295
...
if comando == "STOP":
    request = {'op': 'STOP', 'number': encrypt_intvalue(cipherkey, number),
              'attempts': encrypt_intvalue(cipherkey, jogadas)}
    response = sendrecv_dict(client_sock, request)

    if validate_response(client_sock, response):
        emUso = False
        continue
    else:
        print("Erro: resposta do servidor não é válida")
return None
...

```

Figura 2.26: Caso em que é inserido o comando STOP.

Se o comando executado for o comando STOP, é enviada uma mensagem que será posteriormente enviada ao servidor com o valor criptografado do último número jogado e do número total de jogadas (recorrendo à função "encrypt_intvalue()"). Nas linhas 287-289 o jogo é terminado com sucesso se a

resposta do servidor for válida e é atribuída à variável "emUso"o valor False. Caso a resposta do servidor não seja válida, é exibida a respetiva mensagem de erro - linhas 291-292.

2.2.2 Segurança

Encriptação

```
17 ...
18 def encrypt_intvalue(cipherkey, data):
19     cipher = AES.new(cipherkey, AES.MODE_ECB)
20     data2 = cipher.encrypt(bytes("%16d" % (data), 'utf8'))
21     data_tosend = str(base64.b64encode(data2), 'utf8')
22     return data_tosend
23 ...
```

Figura 2.27: Função para encriptar valores a enviar em formato JSON com codificação base64.

Cada número inteiro comunicado entre o servidor e o cliente é encriptado por blocos usando a função AES-128 no modo ECB. A encriptação é realizada do seguinte modo:

1. Conversão do inteiro numa string binária de 128 bits já que a chave de cifragem é passada como argumento da função.

```
88 ...
89 cipherkey = os.urandom(16)
90 cipherkey_toSend = str(base64.b64encode(cipherkey), 'utf8')
91 ...
```

Figura 2.28: Chave de cifragem gerada e codificada no modelo Base64 para ser enviada ao servidor.

2. Codificação no formato Base64 para que os criptogramas sejam suportados pelo JSON;
3. A função "encrypt_intvalue"devolve o valor que foi codificado e encriptado para que possa ser enviado para o servidor.

Descrição

```
17 ...
18 def encrypt_intvalue(cipherkey, data):
19     cipher = AES.new(cipherkey, AES.MODE_ECB)
20     data2 = cipher.encrypt(bytes("%16d" % (data), 'utf8'))
21     data_tosend = str(base64.b64encode(data2), 'utf8')
22     return data_tosend
23 ...
```

Figura 2.29: Função para descriptar valores recebidos em formato JSON com codificação base64.

Cada número inteiro comunicado entre o servidor e o cliente é descriptado por blocos usando a função AES-128 em modo ECB. A descriptação ocorre do seguinte modo:

1. Decodificação dos dados passados à função "decrypt_intvalue" como argumento no formato Base 64 já que a chave de cifragem é passada como argumento da função;

```
88 ...
89 cipherkey = os.urandom(16)
90 cipherkey_tosend = str(base64.b64encode(cipherkey), 'utf8')
91 ...
```

Figura 2.30: Chave de cifragem gerada e codificada no modelo Base64 para ser enviada ao servidor.

2. Descriptação do conteúdo dos dados;
3. Codificação num valor inteiro e a sua devolução por parte da função.

2.2.3 Main

A função main do programa client é responsável por verificar se no momento em que o programa é executado, os argumentos são passados para a linha de comandos de forma correta. O programa pode ser executado de dois modos:

- Com três argumentos: ID, Porto e Máquina;
- Com dois argumentos: ID e Porto.

No caso em que a máquina está omitida nos argumentos deve-se considerar a máquina local(127.0.0.1). A variável "instead2" controla os modos de execução do programa. Quando instead2 == True está a funcionar em modo de dois argumentos com a máquina local automaticamente designada como destino.

Vamos analisar o código desta função:

```

308 ...
309 def main():
310     instead2 = False
311     if len(sys.argv) != 4:
312         if len(sys.argv) == 3:
313             instead2 = True
314         else:
315             print("Deve passar como argumentos: client_id, Porto, DNS")
316             sys.exit(1)
317     try:
318         int(sys.argv[2])
319     except ValueError:
320         print("Porto deve ser um numero inteiro")
321         sys.exit(2)
322     if int(sys.argv[2]) < 0:
323         print("Porto deve ser um numero inteiro positivo")
324         sys.exit(2)
325     if instead2:
326         maquina = "127.0.0.1".split(".")
327     else:
328         maquina = sys.argv[3].split(".")
329     if len(maquina) != 4:
330         print("maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255")
331         sys.exit(2)
332 ...

```

Figura 2.31: Função main

- Nas linhas 310-316 é verificado se são passados três argumentos. Se tal não se verificar é verificado se foram passados dois argumentos. Neste caso, "instead" recebe o valor True. Caso contrário, o programa encerra com uma mensagem a especificar como deve ser feita a execução;
- Nas linhas 317-321 verifica-se se o argumento do Porto é um número inteiro ao tentar converter a sua string para int. Se a conversão falhar, o programa termina com uma mensagem de erro. E, nas linhas 322-324 é averiguado se o argumento do Porto é um número inteiro positivo;
- De seguida nas linhas 325-328 a máquina designada é a máquina local 127.0.0.1. caso o programa esteja a correr em modo de dois argumentos. Se isto não se verificar, é atribuída à variável o valor passado na linha de comandos e a string da máquina é separada num array de strings para que possam ser analisados os seus elementos separadamente;
- Nas linhas 329-331 é verificado se o array dos elementos da maquina possui quatro elementos. Se tal não acontecer é encerrado o programa com uma mensagem de erro;

```

337     ...
338     for digito in maquina:
339         try:
340             int(digito)
341         except ValueError:
342             print("maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255")
343             sys.exit(2)
344         if int(digito) > 255 or int(digito) < 0:
345             print("maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255")
346             sys.exit(2)
347     ...

```

Figura 2.32: Função main

Aqui é verificado para cada elemento do array se são números inteiros ao tentar fazer a conversão de string para int. Verifica-se também se os elementos estão dentro do intervalo válido, entre 0 e 255. Se algum dos casos não se verificar é reportada uma mensagem de erro.

```

351     ...
352     port = int(sys.argv[2])
353
354     if instead2:
355         hostname = "127.0.0.1"
356     else:
357         hostname = sys.argv[3]
358     ...

```

Figura 2.33: Função main

Por fim, tendo já verificado a validade dos argumentos passados na linha de comandos, são atribuídos às suas respectivas variáveis de acordo com o modo de execução, com dois ou três argumentos.

2.3 Testes unitários e funcionais

2.3.1 Testes Funcionais

Para assegurar que os programas, cliente e servidor, desenvolvidos durante este projeto estão a funcionar corretamente, de acordo com aquilo que é esperado pelos desenvolvedores, implementamos uma série de testes funcionais que se encarregam de verificar a robustez do código previamente descrito no capítulo de metodologia deste relatório.

A intenção por trás da inclusão dos testes é nos certificarmos de que, quanto à inicialização de ambos os programas com a passagem de argumentos na linha de comandos, o código é capaz de identificar as falhas neste processo, comunicá-las ao utilizador (para que este possa inserir argumentos corretos na próxima tentativa de execução), e encerrar o programa corretamente, sem maiores im-

pactos negativos que poderiam advir da passagem de argumentos inválidos, ou insuficientes ao sistema.

Para implementar testes funcionais em Python, primeiro instalamos e importamos o pacote ‘pytest’, que suportará o desenvolvimento dos mesmos, em conjunto com os pacotes ‘Popen’ e ‘PIPE’ advindos de ‘subprocess’, num novo ficheiro chamado ‘funcTests.py’.

Dentro deste ficheiro, definimos duas funções, as quais representam os dois macro-cenários de possíveis erros durante a execução inicial dos programas: ‘numArgs()’, que trata de verificar a robustez dos programas quando lidando com argumentos a menos, ou a mais, do que o esperado para suas respectivas execuções; e ‘invalidArgs()’, que tal como seu nome diz, irá verificar a robustez dos programas quando lidando com argumentos em formatos inválidos, ou seja, diferentes dos esperados pelo sistema ao processá-los.

```
9 def numArgs():
10     proc = Popen("python3 client.py", stdout=PIPE, shell=True)
11     assert proc.wait() == 1
12     assert proc.stdout.read().decode('utf-8') == "Deve passar como argumentos: client_id, Porto, DNS\n"
13
14     proc = Popen("python3 client.py mauro", stdout=PIPE, shell=True)
15     assert proc.wait() == 1
16     assert proc.stdout.read().decode('utf-8') == "Deve passar como argumentos: client_id, Porto, DNS\n"
17
18     proc = Popen("python3 client.py mauro 2345 127.0.0.1 67 juju", stdout=PIPE, shell=True)
19     assert proc.wait() == 1
20     assert proc.stdout.read().decode('utf-8') == "Deve passar como argumentos: client_id, Porto, DNS\n"
21
22     print("*- CLIENTE numero de argumentos inválido: PASSOU")
23
24     proc = Popen("python3 server.py", stdout=PIPE, shell=True)
25     assert proc.wait() == 1
26     assert proc.stdout.read().decode('utf-8') == "Deve passar o porto como argumento para o servidor\n"
27
28     proc = Popen("python3 server.py 2344 567 78", stdout=PIPE, shell=True)
29     assert proc.wait() == 1
30     assert proc.stdout.read().decode('utf-8') == "Deve passar o porto como argumento para o servidor\n"
31
32     print("*- SERVIDOR numero de argumentos inválido: PASSOU")
```

Todos os erros relativos ao número de argumentos inválidos implicam que a execução do programa termine, com código ‘1’. Por isso fazemos uma asserção para cada teste, de que o código retornado por ele deve ser também 1. A segunda asserção também implementada para testes dentro desta função, diz respeito à mensagem que é retornada pelo programa, quando identificada a falha. Caso a mensagem retornada, seja aquela que se esperava, então o programa lidou corretamente com as falhas.

```

36 def invalidArgs():
37     proc = Popen("python3 client.py mauro patricia labi", stdout=PIPE, shell=True)
38     assert proc.wait() == 2
39     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro\n"
40
41     proc = Popen("python3 client.py mauro -5 127.0.0.1", stdout=PIPE, shell=True)
42     assert proc.wait() == 2
43     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro positivo\n"
44
45     proc = Popen("python3 client.py mauro 27,4 127.0.0.1", stdout=PIPE, shell=True)
46     assert proc.wait() == 2
47     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro\n"
48
49     proc = Popen("python3 client.py mauro 3456 lol", stdout=PIPE, shell=True)
50     assert proc.wait() == 2
51     assert proc.stdout.read().decode('utf-8') == "maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255\n"
52
53     proc = Popen("python3 client.py mauro 3456 256.0.0.1", stdout=PIPE, shell=True)
54     assert proc.wait() == 2
55     assert proc.stdout.read().decode('utf-8') == "maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255\n"
56
57     print("*- CLIENTE tipo dos argumentos invalidos: PASSOU")
58
59     proc = Popen("python3 server.py -23", stdout=PIPE, shell=True)
60     assert proc.wait() == 2
61     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro positivo\n"
62
63     proc = Popen("python3 server.py banana", stdout=PIPE, shell=True)
64     assert proc.wait() == 2
65     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro\n"
66
67     print("*- SERVIDOR tipo dos argumentos invalidos: PASSOU")

```

Todos os erros relativos a tipos de argumentos inválidos implicam que a execução do programa termine, com código '2'. Por isso fazemos uma asserção para cada teste, de que o código retornado por ele deve ser também 2. A segunda asserção também implementada para testes dentro desta função, diz respeito à mensagem que é retornada pelo programa, quando identificada a falha. Caso a mensagem retornada, seja aquela que se esperava, então o programa lidou corretamente com as falhas.

2.3.2 Teste unitário

Para além dos testes funcionais, foi também desenvolvido, num ficheiro chamado 'unitTests.py', um simples teste unitário que vai testar as funções responsáveis por criptografar, e descriptografar, valores inteiros recebidos e enviados pelo programa cliente. Para que o teste seja eficaz, utilizamos novamente o pacote 'pytest'. Desta vez, para realizar o teste com precisão, também utilizamos os pacotes 'os' e 'base64', que irão gerar uma chave de cifra exatamente como aquela utilizada nos programas em que as funções aparecem.

```

36 def invalidArgs():
37     proc = Popen("python3 client.py mauro patricia labi", stdout=PIPE, shell=True)
38     assert proc.wait() == 2
39     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro\n"
40
41     proc = Popen("python3 client.py mauro -5 127.0.0.1", stdout=PIPE, shell=True)
42     assert proc.wait() == 2
43     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro positivo\n"
44
45     proc = Popen("python3 client.py mauro 27,4 127.0.0.1", stdout=PIPE, shell=True)
46     assert proc.wait() == 2
47     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro\n"
48
49     proc = Popen("python3 client.py mauro 3456 lol", stdout=PIPE, shell=True)
50     assert proc.wait() == 2
51     assert proc.stdout.read().decode('utf-8') == "maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255\n"
52
53     proc = Popen("python3 client.py mauro 3456 256.0.0.1", stdout=PIPE, shell=True)
54     assert proc.wait() == 2
55     assert proc.stdout.read().decode('utf-8') == "maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255\n"
56
57     print("*- CLIENTE tipo dos argumentos invalidos: PASSOU")
58
59     proc = Popen("python3 server.py -23", stdout=PIPE, shell=True)
60     assert proc.wait() == 2
61     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro positivo\n"
62
63     proc = Popen("python3 server.py banana", stdout=PIPE, shell=True)
64     assert proc.wait() == 2
65     assert proc.stdout.read().decode('utf-8') == "Porto deve ser um numero inteiro\n"
66
67     print("*- SERVIDOR tipo dos argumentos invalidos: PASSOU")

```

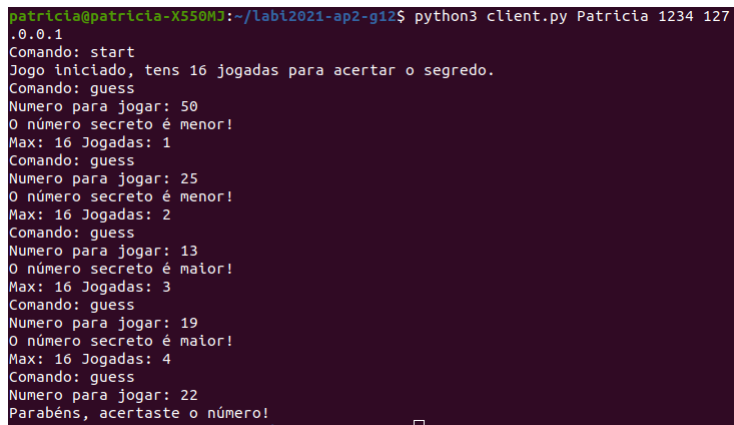
Para que as funções passem este teste, encriptamos um número 1212, e depois o descriptografamos, recorrendo às funções aqui analisadas, e verificamos que o número retornado por elas após o processo de criptogra é o mesmo. O que prova que as funções estão a funcionar corretamente, ou não.

Capítulo 3

Resultados/Análise

Neste capítulo será apresentada a interação via terminal entre um ou mais clientes e o servidor.

- Jogo com um cliente e o servidor



```
patricia@patricia-X550M3:~/lab12021-ap2-g12$ python3 client.py Patricia 1234 127.0.0.1
Comando: start
Jogo iniciado, tens 16 jogadas para acertar o segredo.
Comando: guess
Numero para jogar: 50
O número secreto é menor!
Max: 16 Jogadas: 1
Comando: guess
Numero para jogar: 25
O número secreto é menor!
Max: 16 Jogadas: 2
Comando: guess
Numero para jogar: 13
O número secreto é maior!
Max: 16 Jogadas: 3
Comando: guess
Numero para jogar: 19
O número secreto é maior!
Max: 16 Jogadas: 4
Comando: guess
Numero para jogar: 22
Parabéns, acertaste o número!
```

Figura 3.1: Terminal do cliente.

A aplicação está a funcionar de modo correto. É visível que quando o comando START é introduzido a aplicação indica que o jogo foi iniciado e exibe o número de tentativas que o cliente tem disponível para adivinhar o número secreto. Depois, sempre que se introduz o comando GUESS, é pedido o palpite e, após a introdução, é exibido se o número introduzido é maior, menor ou igual ao número secreto. E, também é mostrado o número de tentativas que ainda restam ao cliente para adivinhar o número secreto.

```

patricia@patricia-X550MJ:~$ cd labi2021-ap2-g12
patricia@patricia-X550MJ:~/labi2021-ap2-g12$ python3 server.py 1234
{'op': 'START', 'client_id': 'Patricia', 'cipherkey': 'VQNPAdC1HK+rOC8PAUAPvg=='}
}
{'name': ['Patricia'], 'sock_id': [40764], 'segredo': [22], 'max': [16], 'jogadas': [0], 'resultado': [], 'cipherkey': [b'U\x030\x01\xd0\xb5\x1c\xaf\xab8/\x0f\x01@\x0f\xbe']}
{'op': 'GUESS', 'number': 'I/ufnPpaB1oBKqyr4PQnPQ=='}
b' 50'
{'op': 'GUESS', 'number': 'yeBHtrGLS6s/Kj4j6q5QLQ=='}
b' 25'
{'op': 'GUESS', 'number': '+gow0YymJmYLD51yM0kOPg=='}
b' 13'
{'op': 'GUESS', 'number': '635JtsP1l2GXJAWxd3JlLQ=='}
b' 19'
{'op': 'GUESS', 'number': 'uVDSaeHLCKmvtIHLn+ezJw=='}
b' 22'
{'op': 'STOP', 'number': 'uVDSaeHLCKmvtIHLn+ezJw==', 'attempts': 'tyGIPvGPNfNgecog4dHuuQ=='}
b' 5'
b' 22'
numero de gamers: 1
index: 0
CURRENT GAMERS: {'name': [], 'sock_id': [], 'segredo': [], 'max': [], 'jogadas':

```

Figura 3.2: Terminal do servidor.

O servidor está a acompanhar cada etapa executada pelo cliente. Por exemplo, quando o cliente introduziu o comando START é armazenado no dicionário os dados dos jogadores que estão ativos no jogo. Neste caso, o cliente Patrícia com o "sock_id": 40764, o número secreto : 22, o número máximo de jogadas: 16, o número de jogadas efetuadas, o resultado e a chave de cifragem.

- Operação QUIT

```

patricia@patricia-X550MJ:~/labi2021-ap2-g12$ python3 client.py Patricia 1234 127.0.0.1
Comando: start
Jogo iniciado, tens 17 jogadas para acertar o segredo.
Comando: guess
Numero para jogar: 50
O número secreto é maior!
Max: 17 Jogadas: 1
Comando: guess
Numero para jogar: 75
O número secreto é menor!
Max: 17 Jogadas: 2
Comando: quit
Jogo terminado com sucesso.

```

Figura 3.3: Terminal do cliente.

A operação QUIT foi bem sucedida. Após a introdução do comando QUIT a mensagem exibida foi "Jogo terminado com sucesso"o que indica a desistência do cliente pois o número de tentativas ainda não tinha sido esgotado.


```
{'op': 'QUIT'}
numero de gamers: 1
index: 0
CURRENT GAMERS: {'name': [], 'sock_id': [], 'segredo': [], 'max': [], 'jogadas':
[], 'resultado': [], 'cipherkey': []}
```

Figura 3.4: Terminal do servidor.

Após a operação QUIT ser executada o cliente é removido da lista de jogadores ativos. Assim, o dicionário encontra-se vazio pois não há nenhum cliente com um jogo iniciado.

- Clientes com o mesmo client_id

```
patricia@patricia-X550M3:~/lab12021-ap2-g12$ python3 client.py Patricia 1234 127
.0.0.1
Comando: start
Erro: Cliente existente
Comando: █
```

Figura 3.5: Ação da aplicação quando existem dois clientes com o mesmo id.

Havendo dois clientes com o mesmo client_id a jogar ao mesmo tempo é de esperar que a aplicação exiba um erro a informar que já existe um cliente com esse id. Conclui-se que o programa está a funcionar corretamente pois é imprimida a mensagem "Erro: Cliente existente".

- Clientes diferentes a jogar

```
Comando: start                                     patricia@patricia-X550M3:~/lab12021-ap2-g12$ python3 client.py Patricia 1234 127
Jogo iniciado, tens 15 jogadas para acertar o segredo. .0.0.1
Comando: guess                                     Comando: start
Comando: guess                                     Jogo iniciado, tens 16 jogadas para acertar o segredo.
Numero para jogar: 58                               Comando: guess
O número secreto é menor!                           Numero para jogar: 58
Max: 15 Jogadas: 1                                  O número secreto é menor!
Comando: 25                                         Max: 16 Jogadas: 1
Comando: guess                                     Comando: guess
Numero para jogar: 25                               Numero para jogar: 25
O número secreto é maior!                           O número secreto é menor!
Max: 15 Jogadas: 2                                  Max: 16 Jogadas: 2
Comando: guess                                     Comando: guess
Numero para jogar: 37                               Numero para jogar: 12
O número secreto é maior!                           O número secreto é maior!
Max: 15 Jogadas: 3                                  Max: 16 Jogadas: 3
Comando: guess                                     Comando: guess
Numero para jogar: 44                               Numero para jogar: 17
O número secreto é menor!                           O número secreto é menor!
Max: 15 Jogadas: 4                                  Max: 16 Jogadas: 4
Comando: guess                                     Comando: guess
Numero para jogar: 40                               Numero para jogar: 14
O número secreto é menor!                           Parabéns, acertaste o número!
Max: 15 Jogadas: 5                                  patricia@patricia-X550M3:~/lab12021-ap2-g12$ █
Comando: guess
Numero para jogar: 38
Parabéns, acertaste o número!
```

Figura 3.6: Ação da aplicação quando existem dois clientes diferentes a jogar simultaneamente.

Neste caso, como o client_id dos dois jogadores ativos é diferente a aplicação funciona corretamente, como seria de esperar, não havendo conflitos nem ocorrência de erros.

- Introdução de comandos incorrectos

```

Numero para jogar: 14
Parabéns, acertaste o número!
patricia@patricia-X550M3:~/labi2021-ap2-g12$ python3 client.py Patricia 1234 127.0.0.1
Comando: start
Jogo iniciado, tens 30 jogadas para acertar o segredo.
Comando: guess
Numero para jogar: abc
Jogada deve ser um numero
Numero para jogar: 500
Numero deve estar entre 0 e 100
Numero para jogar: 50
O número secreto é maior!
Max: 30 Jogadas: 1
Comando: abc
Comando: 123
Comando: guess
Numero para jogar: 76
O número secreto é menor!
Max: 30 Jogadas: 2
Comando: quit
patricia@patricia-X550M3:~/labi2021-ap2-g12$ python3 client.py Patricia 1234 547.0.0.1
maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255
patricia@patricia-X550M3:~/labi2021-ap2-g12$

```

Figura 3.7: Comportamento da aplicação perante introdução de comandos incorrectos.

Após a introdução de uma string em vez de um número é exibida a mensagem "Jogada deve ser um numero".E, após a introdução de um número superior ao intervalo é imprido que o número deve estar entre 0 e 100. Para além disso, quando se coloca o nome/endereço da máquina incorretamente aparece a mensagem de erro "Maquina deve ser especificada no formato x.x.x.x, com x entre 0 e 255".

- Armazenamento dos dados no ficheiro report.csv

1	name	sock_id	segredo	max	jogadas	resultado
2	Patricia	40764	22	16	5	SUCCESS
3	Patricia	40772	56	17	2	DESISTENCIA
4	Mauro	40786	38	15	6	SUCCESS
5	Patricia	40778	14	16	5	SUCCESS
6	Patricia	40792	67	30	2	DESISTENCIA

Figura 3.8: Ficheiro report.csv

O ficheiro report.csv foi atualizado corretamente com os dados dos jogadores ativos no jogo.

Capítulo 4

Conclusões

Durante o desenvolvimento deste projeto, pudemos colocar em prática vários conhecimentos que nos foram apresentados em aula durante o semestre, tais como: sockets, comunicação de mensagens TCP entre servidores e clientes, gravação de dados em ficheiros .CSV, criptografia, testes e depuração do código, entre outros mais.

O nosso objetivo com este projeto é desenvolver um jogo de adivinha o número, onde diversos clientes se podem conectar a um servidor que gerencia os jogos e armazena dados num cheiro .CSV. Este pode ser considerado um jogo simples em conceito, mas que requer esforço e dedicação para ser implementado de forma correta, clara e robusta. Os programas "client.py" e "server.py", acompanhados do pacote "common_comm.py", formam juntos um jogo completo e totalmente funcional.

O cliente é capaz de gerenciar inputs do utilizador e executar pedidos ao servidor corretamente, recorrendo às suas várias funções anteriormente descritas, tendo a certeza de que todos os dados numéricos inteiros estão seguros por uma criptografia de ponta a ponta, que foi testada graças à implementação de um teste unitário. O servidor por sua vez, recebe os pedidos dos vários possíveis clientes a jogar em simultâneo e é capaz de processá-los e devolver uma resposta ao cliente, criando assim uma interação completa e orgânica entre ambos os programas. Caso algo inesperado ocorra durante o jogo, os programas aqui desenvolvidos são programados para corrigir estas falhas sem consequências negativas. Por fim, a execução do jogo está também testada através de testes funcionais, o que torna o projeto aqui escrito numa completa demonstração de conhecimento e aplicações práticas dos conteúdos desta cadeira.

Acreditamos não somente que atingimos a meta proposta para o trabalho, mas que também a excedemos em termos de aprendizagem prática. Todos os algoritmos, estratégias e motivos anteriormente descritos neste relatório constataam o longo caminho que percorremos para alcançar o resultado final que é aqui apresentado com orgulho.

Contribuições dos autores

Cada aluno fez o equivalente a cinquenta por cento.