Trabalho de aprofundamento

Objetivos:

- Planeamento e preparação do trabalho de aprofundamento.

Este guião apresenta as regras do trabalho de aprofundamento.

1.1 Regras

O trabalho deve ser realizado por um grupo de 2 alunos e entregue, via a plataforma `https://elearning.ua.pt`, dentro do prazo lá indicado. A entrega deverá ser feita por **apenas um dos membros** do grupo e deve consistir de **um único arquivo** `.zip`, `.tgz` (TAR comprimido por `gzip`) ou `.tbz` (TAR comprimido por `bzip2`).

O arquivo deve conter os ficheiros de código (`server.py` e `client.py` e eventuais ficheiros de testes). Deve também conter um ficheiro de texto simples designado por `readme.txt` onde deve identificar os elementos do grupo (nome e número mecanográfico) e indicar a percentagem de trabalho realizada por cada aluno do grupo. Esta auto-avaliação afetará a ponderação da nota a atribuir a cada elemento.

Só serão avaliados trabalhos enviados via a plataforma `https://elearning.ua.pt`. Ficheiros corrompidos ou inválidos não serão avaliados à posteriori e não será permitido o reenvio. Verifique bem os ficheiros colocados no arquivo antes de submeter o trabalho.

1.2 Avaliação

A avaliação irá incidir sobre:

1. **cumprimento dos requisitos apresentados**,
2. **qualidade do código produzido e comentários**,
3. **testes unitários e funcionais realizados**,
4. **o suporte de segurança adicionado**.

1.3 Tema Proposto

O objetivo deste trabalho é criar um servidor que suporte a análise estatística de listas de números inteiros. Mais concretamente que determine, o mínimo e o máximo números de uma lista de números inteiros fornecidos pelos clientes.

O servidor nunca deverá aceitar dois clientes com a mesma identificação a requisitar simultaneamente o serviço e deverá criar e atualizar um ficheiro designado por `report.csv` onde vai escrevendo os dados e resultados dos diversos clientes quando estes terminam o pedido do serviço. O cliente pode desistir em qualquer altura e a análise fica sem efeito. Neste caso a informação relativa ao serviço não fica registada.

Quando o cliente indicar ao servidor que o envio de dados acabou e receber os dados da análise estatística, ele deve escrever no monitor uma mensagem apresentando os números enviados ao servidor e os valores mínimo e máximo recebidos do servidor. Por sua vez o servidor acrescenta ao ficheiro a informação relativa ao cliente: cliente; número de números recebidos; número mínimo e número máximo.

1.4 Preparação do trabalho

1. Forme um grupo de trabalho de 2 alunos e inscreva-o na **Pauta de Formação de Grupos para o Trabalho de Aprofundamento**, disponibilizada na página da disciplina no *elearning*.

2. Descarregue o **Arquivo de ficheiros python**, disponibilizado também na página da disciplina no *elearning*, e que contém todos os programas necessários para a compreensão e realização do trabalho.

3. Crie ou utilize um projeto que já tenha criado no CodeUA e utilize-o para agilizar a interação dos elementos do grupo durante a realização do trabalho. Não se esqueça de utilizar um repositório **git** e de o actualizar com regularidade de forma a manter as sucessivas versões do trabalho. E de maneira a não perder o seu trabalho se tiver algum problema com o seu computador pessoal.

1.5 Protocolo utilizado

A comunicação entre os clientes e o servidor deverá ser suportada por *sockets* TCP, porque estes têm a vantagem de permitir detetar a falha de um interlocutor aquando do uso do *socket* com a ligação para com o mesmo.

O cliente possui os seguintes requisitos funcionais:

1. Deve ser invocado no formato: `python3 client.py client_id porto [máquina]`;
2. O segundo argumento é o identificador pessoal do cliente que pretende aceder ao servidor;
3. O terceiro argumento deverá ser um valor inteiro positivo, especificando o porto TCP do servidor;
4. O quarto argumento deverá ser um nome DNS ou endereço IPv4 no formato X.X.X.X, onde X representa um valor inteiro decimal entre 0 e 255. Se este argumento não for indicado, o cliente deverá usar um servidor na mesma máquina onde está (`localhost`);
5. Se for iniciado com um número incorreto de argumentos ou com qualquer argumento inválido (tipo errado, valor incorreto ou não encontrado), deverá ser apresentada uma mensagem de erro respetiva ao argumento que gera o erro.

O servidor possui os seguintes requisitos funcionais:

1. Deve ser invocado no formato: `python3 server.py porto`;
2. Se for iniciado com um número incorreto de argumentos ou com qualquer argumento inválido (tipo errado, valor incorreto ou não encontrado), deverá ser apresentada uma mensagem de erro respetiva ao argumento que gera o erro.

O ficheiro `report.csv` deverá possuir a seguinte estrutura: identificador do cliente, número de números recebidos do cliente, e os resultados (número mínimo e número máximo).

Aconselha-se também que as mensagens sejam estruturadas em JSON, porque elas podem ser facilmente convertidas de e para dicionários Python. De seguida vamos descrever as operações que devem ser implementadas e o formato das respectivas mensagens.

1.6 Descrição do modo de funcionamento do serviço

O processo deve funcionar da seguinte forma:

1. O cliente contacta o servidor para dar início ao serviço (operação **START**) enviando um dicionário com o seguinte formato:

   ```
   { "op": "START", "client_id": nome identificador do cliente }
   ```

   O servidor só deverá aceitar o cliente caso ele não esteja já inscrito na lista de clientes ativos do servidor e acrescenta-o à sua estrutura de dados de clientes ativos (para posteriormente evitar a inscrição de outro cliente com a mesma identificação). Devolve um dicionário indicando que a operação de registo deste cliente foi feita com sucesso:

   ```
   { "op": "START", "status": True }
   ```

   Se pelo contrário um cliente com o mesmo identificador já se encontra registado no servidor então devolve-lhe um dicionário indicando que a operação de registo deste cliente não teve sucesso com o seguinte formato:

   ```
   { "op": "START", "status": False, "error": "Cliente existente" }
   ```

2. O cliente contacta o servidor para desistir (operação **QUIT**) enviando um dicionário com o seguinte formato: `{ "op": "QUIT" }`

   O servidor só deverá aceitar a desistência do cliente caso ele esteja presentemente ativo. Nesse caso remove o registo de cliente da sua estrutura de dados de clientes ativos (para posteriormente aceitar a inscrição de outro cliente com a mesma identificação) e devolve um dicionário indicando que a operação de desistência deste cliente foi feita com sucesso com o seguinte formato:

   ```
   { "op": "QUIT", "status": True }
   ```

   Se pelo contrário o cliente não está registado no servidor então devolve-lhe um dicionário indicando que a operação de desistência deste cliente não teve sucesso com o seguinte formato:

   ```
   { "op": "QUIT", "status": False, "error": "Cliente inexistente" }
   ```

3. O cliente contacta o servidor para enviar um número (operação **NUMBER**) enviando um dicionário com o seguinte formato:

   ```
   { "op": "NUMBER", "number": valor numérico inteiro }
   ```

   O servidor só deverá aceitar os dados do cliente caso ele esteja presentemente ativo. Nesse caso contabiliza a receção de mais um valor recebido do cliente e armazena-o internamente. O servidor envia ao cliente um dicionário com o seguinte formato:

   ```
   { "op": "NUMBER", "status": True }
   ```

   Se pelo contrário o cliente não está registado no servidor então devolve-lhe um dicionário indicando que a operação de envio de dados deste cliente não teve sucesso com o seguinte formato:

   ```
   { "op": "NUMBER", "status": False, "error": "Cliente inexistente" }
   ```

4. O cliente contacta o servidor para terminar o envio de dados (operação **STOP**) enviando um dicionário com o seguinte formato:

   ```
   { "op": "STOP" }
   ```

   O servidor só deverá aceitar a terminação de envio de dados do cliente caso ele esteja presentemente ativo. O servidor deve verificar se o cliente lhe enviou dados e assim sendo determina o resultado (número mínimo e número máximo) e atualiza o ficheiro `report.csv` com os dados do cliente e o resultado do serviço. Senão ignora o serviço e nada regista no ficheiro de resultados. O servidor remove o registo de cliente da sua estrutura de dados de clientes ativos (para posteriormente aceitar a inscrição de outro cliente com a mesma identificação). Caso o serviço tenha sido efetivamente executado devolve um dicionário, indicando que a operação foi feita com sucesso e o resultado da análise estatística, com o seguinte formato:

   ```
   { "op": "STOP", "status": True, "min": número mínimo, "max": número máximo }
   ```

   Quando o cliente recebe esta informação do servidor escreve no monitor uma mensagem a indicar os números enviados ao servidor e os resultados recebidos do servidor.

   Se pelo contrário o cliente não está registado no servidor ou não lhe enviou quaisquer dados, então o servidor devolve-lhe um dicionário indicando que a operação de terminação não teve sucesso e indicando o erro de "Cliente inexistente" ou de "Dados insuficientes", com o seguinte formato:

   ```
   { "op": "STOP", "status": False, "error": um dos erros indicados em cima }
   ```

1.7 Troca de dicionários Python via *sockets* TCP

Tal como referimos anteriormente as mensagens devem ser estruturadas em JSON porque elas podem ser facilmente convertidas de e para dicionários Python.

Para fazer a comunicação entre o cliente e o servidor deve usar o módulo **`comon_comm.py`** (ver Figura 1.1). Este módulo disponibiliza três operações de alto nível que encapsulam e desencapsulam os pedidos e as respostas (no formato de dicionários Python) que são trocados entre o cliente e o servidor para executar a funcionalidade da aplicação.

As operações **`recv_dict`** e **`send_dict`** devem ser usadas pelo servidor para, respectivamente, receber o pedido da operação (**START**, **NUMBER**, **STOP** e **QUIT**) que o cliente pretende executar e para enviar-lhe a respetiva resposta com o resultado de execução da mesma. Por sua vez a operação **`sendrecv_dict`** deve ser usada pelo cliente para enviar um pedido de execução de uma operação aguardando pela respetiva resposta do servidor.

```
import socket
import json
import base64

#
# Universal function to send a given amount of data to a TCP socket.
# It returns True or False, depending on the success of
# sending all the data to the socket.
#
def exact_send (dst, data):
    try:
        while len (data) != 0:
            bytes_sent = dst.send (data)
            data = data[bytes_sent : ]
        return True
    except OSError:
        return False

#
# Universal function to receive a given amount of data from a TCP socket.
# It returns None or data, depending on the success of
# receiving all the required data from the socket.
#
def exact_recv (src, count):
    data = bytearray (0)
    while count != 0:
        new_data = src.recv (count)

        if len (new_data) == 0: return None

        data += new_data
        count -= len (new_data)

    return data
```

```python
#
# Universal function to send a dictionary message to a TCP socket.
# It actually transmits a JSON object, prefixed by its length (in network byte order).
#
# The JSON object is created from the dictionary.
# It returns True or False, depending on the success of sending the
# message to the socket.
#
def send_dict (dst, msg):
    # DEBUG print ("Send: %s" % (msg))
    data = bytes(json.dumps (msg), "utf8")
    prefixed_data = len (data).to_bytes (4, "big") + data
    return exact_send (dst, prefixed_data)

#
# Universal function to receive a dictionary message from a TCP socket.
# It actually receives a JSON object, prefixed by its length (in network byte order).
# The dictionary is created from that JSON object.
#
def recv_dict (src):
    prefix = exact_recv (src, 4)

    if prefix == None: return None

    length = int.from_bytes (prefix, "big")
    data = exact_recv (src, length)

    if data == None: return None

    msg = json.loads (str(data, "utf8"))
    # DEBUG print ("Recv: %s" % (msg))
    return msg

#
# Universal function to send and receive a dictionary to/from a TCP socket peer.
# It returns None upon an error.
#
def sendrecv_dict (peer, msg):
    if send_dict (peer, msg):
        return recv_dict (peer)
    else:
        return None
```

Figura 1.1: Módulo de comunicações

1.8 Interação do servidor com os clientes

A Figura 1.2 apresenta a interação do servidor com os clientes. Para que um servidor possa atender vários clientes ele necessita de ter uma lista para armazenar os *sockets* associados aos clientes ativos. O método **select** permite ficar à escuta de informação de múltiplas fontes, indicando depois qual das fontes possui informação para ser consumida. Basicamente o método permite monitorizar *sockets*.

```
. . .
server_socket = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind (("127.0.0.1", port))
server_socket.listen ()
. . .
clients = []
. . .
while True:
    try:
        available = select.select ([server_socket] + clients, [], [])[0]
    except ValueError:
        # Sockets may have been closed, check for that
        for client in clients:
            if client.fileno () == -1: clients.remove (client) # closed
        continue # Reiterate select

    for client_sock in available:
        # New client?
        if client_sock is server_socket:
            newclient, addr = server_socket.accept ()
            clients.append (newclient)

        # Or a client message/disconnect?
        else: # See if client sent a message
            if len (client_sock.recv (1, socket.MSG_PEEK)) != 0:
                new_msg (client_sock)    # client socket has a message
            else: # Or just disconnected
                clients.remove (client_sock)
                clean_client (client_sock)
                client_sock.close ()
                break # Reiterate select
. . .
```

Figura 1.2: Interação do servidor com os clientes

Este método **select** aceita três parâmetros: a lista de *sockets* (ou outros dispositivos, como por exemplo, o *stdin*) onde se espera por dados, a lista de *sockets* onde recentemente foram escritos dados e se espera que estes sejam transmitidos e a lista de *sockets* onde se quer receber notificações de excepções (por exemplo, *socket* fechado). E o método devolve igualmente três listas com os *sockets* que tiveram os eventos respetivos.

Aqui apenas nos interessa monitorizar a primeira lista, ou seja os *sockets* associados aos clientes (lista **clients**) e também o *socket* do servidor pois é ele que recebe a notificação de um cliente que quer comunicar com o servidor (método **connect** invocado pelo cliente). Como estamos apenas interessados na primeira lista o método é invocado na forma:

```
select.select ([server_socket] + clients, [], [])[0].
```

Ao invocar o método **select** para monitorizar os *sockets* com informação pronta a ser consumida, o método falha quando algum dos *sockets* foi desligado. Nessa situação de exceção é necessário recorrer ao método **fileno** que devolve -1 como sinal de fracasso na tentativa de contato com o *socket* do cliente, confirmando que ele foi desligado e consequentemente é removido da lista de clientes ativos.

Quando o método **select** indica que há informação válida nos *sockets* analisados podemos ter uma de duas situações possíveis. Ou foi o *socket* do servidor que recebeu dados e nesse caso estamos perante um novo cliente que tem de ser adicionado à lista de clientes ativos. Se pelo contrário foi o *socket* de um cliente temos que verificar se existe mesmo uma mensagem e neste caso é invocada a operação **new_msg** que vai processar o pedido do cliente ou se o cliente foi desligado e é preciso removê-lo da lista de clientes ativos.

Além da lista de *sockets* dos clientes ativos, o servidor também tem um dicionário (designado por **users**) onde armazena todos os dados dos clientes ativos. Cada cliente que requesita a operação **START** caso tenha uma identificação diferente dos clientes ativos é adicionado a este dicionário sendo que a sua chave que o identifica no dicionário é o **client_id** que foi indicado quando executou a aplicação cliente.

Vamos considerar que três clientes (anton, jonas e manel) executaram a aplicação cliente e acionaram a operação **START** (ver Figura 1.3), sendo que apenas o terceiro indicou que vai interagir com o servidor em modo encriptado.

```
{ "op": "START", "client_id": "anton", "cipher": None }
{ "op": "START", "client_id": "jonas", "cipher": None }
{ "op": "START", "client_id": "manel", "cipher": cipherkey em base64 }
```

Figura 1.3: Operações **START** dos clientes anton, jonas e manel

Observação Importante: Na Secção 1.10 abordaremos os aspetos relacionados com a segurança na comunicação dos dados entre os clientes e o servidor.

A Figura 1.4 apresenta o dicionário com a informação relativa aos três clientes. Repare que cada entrada do dicionário é identicado pelo **client_id** e que apenas o cliente manel enviou uma chave de cifra binária de 16 octetos (128 *bits*), porque pretende interagir com o servidor em modo encriptado.

```
users = {
            "anton": { "socket": <socket ... raddr=("127.0.0.1", 46192)>, "cipher": None,
                       "numbers": []  },

            "jonas": { "socket": <socket ... raddr=("127.0.0.1", 46194)>, "cipher": None,
                       "numbers": [] },

            "manel": { "socket": <socket ... raddr=("127.0.0.1", 46196)>,
                       "cipher": b")6\x0c\xba\xf8\x14\xa7iU\xac\x8a~\xe0H~0",
                       "numbers": [] }
        }
```

Figura 1.4: Dicionário com a informação relativa aos clientes ativos

É preciso ter em atenção que à exceção da operação **START** as restantes operações descritas anteriormente na Secção 1.6 não incluem na respetiva mensagem enviada pelo cliente a sua identificação (**`"client_id"`**).

Assim para melhor estruturar a implementação do servidor é conveniente implementar a função **`find_client_id`** que recebe como parâmetro de entrada o *socket* do cliente, pesquisa-o no dicionário de clientes ativos e devolve a respetiva identificação do cliente ou **`None`** caso ele não seja um cliente ativo.

1.9 Resultados armazenados em ficheiro csv

Para melhor estruturar a aplicação servidor também é conveniente implementar uma função para criar o ficheiro `report.csv` só com o cabeçalho **`create_file ()`** e outra função para atualizar o ficheiro com os dados do cliente **`update_file (client_id, result)`**. Deve ser utilizado o módulo `csv` para processar valores tabulados em CSV.

1.10 Segurança

O trabalho pode ser realizado numa de três possibilidades em termos de segurança da informação trocada entre os clientes e o servidor.

1. Os clientes comunicam abertamente com o servidor não havendo encriptamento dos dados. Nesse caso o cliente contacta o servidor para dar início ao serviço enviando um dicionário com o seguinte formato:

   ```
   { "op": "START", "client_id": nome do cliente }
   ```

2. Os clientes comunicam secretamente com o servidor fazendo encriptamento dos dados numéricos. Nesse caso o cliente contacta o servidor para dar início ao serviço enviando um dicionário com o seguinte formato:

   ```
   { "op": "START", "client_id": nome do cliente, "cipher": cipherkey }
   ```

3. Os clientes escolhem se querem comunicar abertamente ou secretamente com o servidor. Nesse caso o cliente contacta o servidor para dar início ao serviço enviando um dicionário com o seguinte formato:

   ```
   { "op": "START", "client_id": nome do cliente, "cipher": None ou cipherkey }
   ```

Caso opte por fazer encriptamento de dados (sistematicamente para todos os clientes ou para os clientes que assim o desejarem), a cifragem dos números inteiros trocados entre o cliente e o servidor deverá ser feita com cifras simétricas por blocos (o uso de cifras contínuas ou de fluxo é desaconselhado porque liberta informação). Nessas cifras não deverá ser usado alinhamento (*padding*) porque o mesmo é inútil.

Porém, cada número inteiro deverá ser guardado num bloco completo a ser processado pela cifra por blocos. Embora a diversidade seja possível, o servidor e todos os clientes deverão usar as mesmas funções de cifra. Recomenda-se o uso de AES-128 que deverá operar em modo ECB (ver Figura 1.5 e Figura 1.6) com a utilização da instrução **`cipher = AES.new (cipherkey, AES.MODE_ECB)`**.

Também na Figura 1.5 e na Figura 1.6 se mostra como se pode cifrar e decifrar um número inteiro imposto por esta função de cifra. Basicamente o número inteiro é convertido numa *string* binária com 128 *bits*, usando o formato **`%16d`** na função **`bytes`**, que depois é cifrada por uma chave de cifra binária aleatória de 16 *bytes* (**`cipherkey = os.urandom(16)`**). Na receção dos dados encriptados e após a decifragem é preciso converter a *string* numérica para inteiro usando a função **`int`**.

Atenção, porém, ao facto de os dicionários suportarem quaisquer valores (associados a chaves textuais), nomeadamente vetores de octetos (*bytes*), o que não é suportado pelo JSON. Este problema coloca-se quando é preciso lidar com criptogramas. Para o resolver, estes valores podem ser guardados nos dicionários usados para estruturar mensagens como valores textuais, usando, por exemplo, o formato Base64.

Ainda na Figura 1.5 e na Figura 1.6 se apresenta como se codifica uma *string* binária para texto usando a função **`base64.b64encode`** e depois se descodifica uma *string* textual para binário usando a função **`base64.b64decode`**, quer para enviar a chave de cifra do cliente para o servidor quer para trocar os valores numéricos cifrados entre o cliente e o servidor e vice-versa.

```python
# encoding=utf-8
import os
import socket
import select
import json
import base64
import random

from common_comm import send_dict, recv_dict, sendrecv_dict
from Crypto.Cipher import AES

def main():
    tcp_s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind (("127.0.0.1", 0))
    # Ligar ao servidor
    tcp_s.connect (("127.0.0.1", 1234))

    cipherkey = os.urandom(16)
    cipherkey_tosend = str (base64.b64encode (cipherkey), "utf8")
    cipher = AES.new (cipherkey, AES.MODE_ECB)

    request = { "op": "START", "cipher": cipherkey_tosend }
    send_dict (tcp_s, request)

    while 1:
        data = int (input ("Valor: "))
        print ("CLIENT - Valor Enviado %d" % (data))
        data = cipher.encrypt (bytes("%16d" % (data), "utf8"))
        data_tosend = str (base64.b64encode (data), "utf8")

        request = { "value": data_tosend }
        response = sendrecv_dict (tcp_s, request)

        data = base64.b64decode (response["value"])
        data = cipher.decrypt (data)
        data = int (str (data, "utf8"))
        print ("CLIENT - Valor Recebido %d" % (data))

    tcp_s.close()

main ()
```

Figura 1.5: Cliente TCP

```python
# encoding=utf-8
import socket
import json
import base64

from common_comm import send_dict, recv_dict, sendrecv_dict
from Crypto.Cipher import AES

def main ():
    tcp_s = socket.socket (socket.AF_INET, socket.SOCK_STREAM)
    tcp_s.bind (("127.0.0.1", 1234))

    tcp_s.listen ()

    # aceitar novos clientes
    client_s, client_addr = tcp_s.accept ()

    request = recv_dict (client_s)
    cipherkey = base64.b64decode (request["cipher"])
    cipher = AES.new (cipherkey, AES.MODE_ECB)

    while 1:
        request = recv_dict (client_s)
        data = base64.b64decode (request["value"])
        data = cipher.decrypt (data)
        data = int (str (data, "utf8"))
        print ("SERVER - Valor Recebido %d" % (data))

        data = data * 10
        print ("SERVER - Valor Enviado %d" % (data))
        data = cipher.encrypt (bytes("%16d" % (data), "utf8"))
        data_tosend = str (base64.b64encode (data), "utf8")
        response = { "value": data_tosend }
        response = send_dict (client_s, response)

    client_s.close ()
    tcp_s.close ()

main ()
```

Figura 1.6: Servidor TCP

Para melhor estruturar as aplicações cliente e servidor é conveniente implementar uma função para encriptar inteiros **`encrypt_intvalue (cipherkey, data)`** e uma função para desencriptar inteiros **`decrypt_intvalue (cipherkey, data)`**.

1.11 Notas importantes / Considerações finais

As aplicações cliente e servidor devem ser robustas, detetar e indicar erros de má utilização por parte dos utilizadores. Assim é importante que as aplicações validem os argumentos, sejam em número quer por defeito quer por excesso, sejam no tipo e correção dos mesmos. Assim vamos considerar que:

- O erro 1 representa insuficiência ou excesso de argumentos passados às aplicações;
- O erro 2 representa um porto inválido;
- O erro 3 representa uma operação incorreta (ou porque não existe ou porque foi pedida quando não devia ser), o que levou a uma resposta com *status* igual a `False` na mensagem de resposta do servidor. Para esta situação e de maneira a estruturar melhor a aplicação cliente também é conveniente implementar uma função. Esta função `validate_response (client_sock, response)` deve imprimir a respetiva mensagem de erro, fechar o *socket* do cliente e terminar a aplicação com o erro 3;
- O erro 4 representa a terminação extemporânea da aplicação cliente. Também para melhor estruturar a aplicação cliente é conveniente implementar uma função `quit_action (client_sock, attempts)` para processar a desistência do cliente. Esta função deve enviar a mensagem **QUIT**, receber a resposta do servidor, escrever no monitor a informação relativa à desistência ou ao eventual erro ocorrido, fechar o *socket* do cliente e terminar a aplicação com o erro 4.

1.12 Sugestões para a realização do trabalho

1. Complete as funções `main` do cliente e do servidor e comente ou implemente a função `create_file` de maneira a fazer uma simulação ainda sem qualquer funcionalidade;
2. Faça testes funcionais às duas aplicações para assegurar que a passagem de argumentos na linha de comandos está correta e robusta;
3. Implemente as operações uma a uma, inicialmente sem fazer encriptamento de dados, pela seguinte ordem: **START**; **QUIT**; **NUMBER**; e **STOP**;
4. Implemente cada operação no cliente e no servidor de maneira a poder testar cada uma delas exaustivamente à medida que vão sendo desenvolvidas. E para cada uma delas implemente as funções auxiliares que foram sugeridas ao longo do enunciado de maneira a estruturar melhor o código;

5. Teste exaustivamente cada operação:
   a) No caso da operação **START** comece por testá-la só com um cliente, depois com dois clientes diferentes, depois com mais de dois clientes e não se esqueça de testar com um cliente repetido, para ver se o cliente repetido é rejeitado;
   b) Teste as restantes operações pela ordem normal e também fora de ordem, nomeadamente antes da operação **START**;
6. Finalmente (**se desejar ter uma valorização do trabalho**) acrescente às operações o encriptamento de dados e teste exaustivamente a solução final.

Glossário

AES	Advanced Encryption Standard (cifra simétrica por blocos)
Base64	Base64 (forma de codificar octetos arbitrários como caracteres)
CSV	Comma Separated Values
DNS	Domain Name System
ECB	Electronic Code Book (modo de cifra por blocos elementar, sem realimentação)
IPv4	Internet Protocol v4
JSON	JavaScript Object Notation
TCP	Transmission Control Protocol (protocolo de transporte da Internet orientado à ligação)