Guião de apoio 1
Introdução aos *sockets* TCP e aos programas cliente-servidor

---

1. Introdução ao tema

Nesta aula, vamos realizar alguns exercícios em *Python* relacionados com *sockets* [1], base fundamental sob a qual os sistemas e aplicações distribuídas são construídos.

Conforme visto na aula TP, o uso de *sockets* para interligação de dois processos requer a chamada de uma série de funções nos processos intervenientes. A Figura 1 resume esse fluxo (ver também os slides da aula TP01 para mais detalhes sobre as funções invocadas).

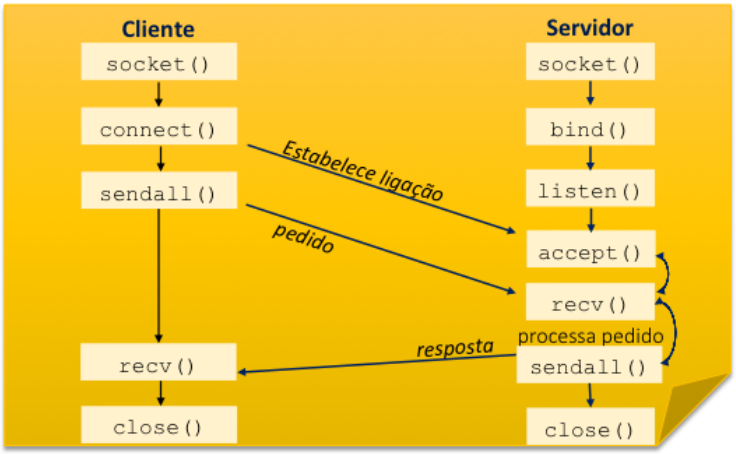

Figura 1. Cliente/servidor com ligação TCP.

Os exemplos a seguir apresentam um cliente que envia uma mensagem a um servidor, o qual apenas imprime essa mensagem e responde ao cliente.

Cliente (*cliente.py*)

```
import socket as s

HOST = '127.0.0.1'
PORT = 9999

sock = s.socket(s.AF_INET, s.SOCK_STREAM)

sock.connect((HOST, PORT))

sock.sendall(b'Vamos aprender isto!')
resposta = sock.recv(1024)

print('Recebi %s' % resposta)

sock.close()
```

Servidor (*servidor.py*)

```
import socket as s

HOST = '' #pode ser vazio, localhost ou 127.0.0.1
PORT = 9999

sock = s.socket(s.AF_INET, s.SOCK_STREAM)
#sock.setsockopt(s.SOL_SOCKET, s.SO_REUSEADDR, 1)
sock.bind((HOST, PORT))
sock.listen(1)
(conn_sock, (addr, port)) = sock.accept()

print('ligado a %s no porto %s' % (addr,port))

msg = conn_sock.recv(1024)
print('recebi %s' % msg)
conn_sock.sendall(b'Aqui vai a resposta')

sock.close()
```

Entre muitos aspetos importantes omitidos para tornar o exemplo simples (alguns dos quais abordaremos a seguir), é de salientar que este código não trata erros.

Num sistema distribuído, os erros podem acontecer a qualquer momento de uma interação, já que falhas parciais podem afetar apenas alguns processos do sistema. Assim, é importante que todas as interações entre processos estejam sempre dentro de blocos *try/except*.

---

2. Exercícios fundamentais

1. Copie os programas cliente e servidor apresentados neste guião e grave-os em ficheiros Python. Execute primeiro o servidor num terminal e imediatamente a seguir o cliente noutro terminal. Qual o output do cliente e do servidor?

2. Modifique os programas para que os endereços e portos a serem utilizados pela socket de escuta do servidor sejam passados pela linha de comando tanto no cliente quanto no servidor. Para tal, é preciso importar o módulo `sys` [2] e utilizar a lista de argumentos passada ao programa (disponível em `sys.argv`).

3. Faça com que a mensagem enviada pelo cliente para o servidor seja lida como input do utilizador. Para tal, é recomendado utilizar o método `input` [3] do próprio Python. Note que é preciso também fazer o encode desta

4. Crie um módulo de nome ***sock_utils.py*** onde irá implementar algumas funções que poderão vir a ser reutilizadas em vários programas. Inicialmente, pretendem-se as seguintes funções:

- **`def create_tcp_server_socket(address, port, queue_size):`**
  Esta função serve para criar uma socket de escuta TCP para servidores, onde poderão posteriormente ser aceites ligações de clientes.
  address será o endereço anfitrião à qual a socket ficará vinculada.
  port será a porta onde o servidor atenderá novos pedidos de ligação.
  queue_size define o número máximo de pedidos de ligação em espera (ver função `listen` dos objetos da classe `socket`).
  Retorna a socket de escuta TCP criada para o servidor.

- **`def create_tcp_client_socket(address, port):`**
  Esta função serve para criar uma socket de ligação para o cliente comunicar com um servidor.
  address será o endereço do servidor onde o cliente se ligará.
  port será a porta onde o servidor atende pedidos de ligação.
  Retorna a socket de ligação que o cliente usará para comunicar com o servidor.

Altere os programas do Exercício 3 de forma a importar este módulo e utilizar as suas funções sempre que for adequado.

5. Modifique o programa servidor para evitar que ele tenha de ser reexecutado cada vez que um cliente envia uma mensagem e recebe uma resposta. Isto pode ser feito, por exemplo, colocando as operações de aceitação e criação da socket de ligação (não a de escuta), receção da mensagem do cliente e envio da resposta dentro de um ciclo.

6. Modifique o programa cliente para evitar que ele tenha de ser reexecutado sempre que queira enviar uma mensagem ao servidor. Isto pode ser feito, por exemplo, colocando as operações de ligação com o servidor, leitura do input, envio da mensagem, receção da resposta e fecho da ligação dentro de um ciclo.

7. Faça com que o programa cliente só saia do ciclo se o utilizador introduzir a *string* "EXIT" (que não deve ser enviada ao servidor).

8. Crie um ficheiro de texto ***mensagens.txt*** com algumas linhas, onde cada linha representa uma mensagem a ser enviada ao servidor. Termine o ficheiro com uma linha somente com a palavra "EXIT". Inicialize o servidor num terminal e utilize noutro terminal este ficheiro para informar o cliente de todas as mensagens que pretende enviar ao servidor, através do seguinte comando:

```
$ cat mensagens.txt | python3 cliente.py localhost 9999
```

Chegaram todas as mensagens ao servidor (exceto o "EXIT")?

---

3. Exercícios complementares

Os alunos que terminem com sucesso os exercícios fundamentais são convidados agora a resolver um exercício adicional:

1. Modifique o módulo ***sock_utils.py*** (Exercício fundamental 4) para incluir a seguinte função:

- **`def receive_all(socket, length):`**
  Esta função receberá exatamente *lenght* bytes através da *socket*. Devem considerar o caso de a socket remota fechar a ligação antes de enviar o número esperado de bytes.
  socket será a socket de ligação para ler os dados.
  lenght determina o número de bytes que devem ser lidos e devolvidos.
  Retorna os bytes recebidos (com tamanho *length*)

2. Modifique o programa dos exercícios fundamentais para que a mensagem enviada pelo cliente seja inserida num dicionário [4] no servidor. A chave deverá ser um número inteiro inicializado a zero, que deverá ser incrementado a cada nova inserção.

3. Modifique o programa anterior para que ele altere o seu comportamento de acordo com a *string* recebida na mensagem do cliente:

- Caso receba a *string* "GET <num>", onde <num> é um número inteiro, retorna a *string* associada à chave <num> no dicionário do servidor, ou a *string* "chave inexistente" caso esse número não seja chave no dicionário.

- Caso receba a *string* "LIST", retorna uma grande *string* com todas as *strings* existentes no dicionário concatenadas e separadas por vírgulas. Se o dicionário estiver vazio, deve retornar a *string* "dicionário vazio".
- Outas *strings* deverão ser inseridas no dicionário e a resposta deve ser uma *string* com a chave criada.

Pergunta: em que medida a concretização dessas operações limita o que podemos guardar na lista?

4. Modifique o programa anterior para evitar que o cliente tenha de ser reexecutado, incluindo a conexão com o servidor, sempre que queira manipular o dicionário no servidor. Isto pode ser feito, por exemplo, colocando as operações de leitura do input, envio da mensagem e receção da resposta dentro de um ciclo. Faça com que o programa só saia do ciclo se o cliente introduzir a *string* "EXIT" (que não deve ser enviada ao servidor).

5. Execute vários programas clientes concorrentemente para manipular uma lista num único servidor. O que observou? Com o que aprendemos até aqui, modifique o programa para que eles efetivamente manipulem a lista em paralelo.

6. Modifique os programas desenvolvidos para inserir uma *string* s no dicionário através da mensagem "ADD,*s*". Adicionalmente, permita que se removam elementos da lista através da mensagem "REMOVE,<num>". Note que com essas modificações o programa passa a não inserir automaticamente qualquer s*tring* recebida, respondendo apenas às mensagens "ADD,*", "REMOVE,*", "GET" e "LIST".

---

4. Bibliografia e outro material de apoio

[1] https://docs.python.org/3/library/socket.html
[2] https://docs.python.org/3/library/sys.html
[3] https://docs.python.org/3/library/functions.html#input
[4] https://docs.python.org/3/tutorial/datastructures.html#dictionaries