Ciências ULisboa

Aplicações Distribuídas 2022/23

Guião de apoio 4
Suporte a múltiplos clientes

---

1. Introdução ao tema

Nos guiões anteriores fizemos com que os servidores suportassem interação continuada com múltiplos clientes fazendo com que cada interação envolvesse a abertura de uma nova ligação, que é fechada logo após a receção da resposta. Isto é ineficiente já que a abertura de uma ligação TCP envolve um *handshake* com três mensagens. Existem duas soluções básicas para essa limitação:

- A primeira solução é a multiplexação de ligações através do uso de uma única linha de execução (*thread*) que monitoriza todas as ligações de clientes com o servidor.
- A segunda é abrir uma linha de execução adicional através da criação de um processo ou uma *thread* para cada ligação aceite no servidor;

Neste guião, começaremos por explorar a primeira solução com o módulo `select`, que se mostra mais adequada para a construção de servidores que requerem pouco processamento por mensagem recebida. De seguida, vamos verificar a segunda alternativa recorrendo ao módulo *socketserver* do Python que foi abordado na aula TP04.

2. Multiplexação de Ligações com o módulo `select`

A construção de servidores capazes de manter múltiplas ligações requer a utilização de primitivas de multiplexação de objetos de *IO* (e.g., *sockets*). Uma das primitivas mais conhecidas e utilizadas para essa finalidade é o *select*, descrito abaixo.

```
import select

R, W, X = select.select(rlist, wlist, xlist, timeout)
```

rlist: lista de objetos monitorizados para leitura
wlist: lista de objetos monitorizados para escrita
xlist: lista de objetos monitorizados para exceções
timeout: se for 0, faz uma monitorização e sai; se for omitido, a função bloqueia até que um dos objetos esteja pronto; valor de virgula flutuante > 0, é um timeout

R, **W** e **X**: Listas com os objetos prontos. São subconjuntos de rlist, wlist e xlist.

Conforme pode ser visto acima, a função *select* permite que um conjunto de *sockets* (mantidos na lista *rlist*) sejam monitorizados. Cada vez que essa função desbloqueia, é porque uma das *sockets* nessa lista tem dados para serem lidos (assumindo que o *timeout* não seja usado).

O código a seguir apresenta um trecho de código de um servidor que atende múltiplos clientes usando a função *select*. O exemplo omite o código inicial até à criação de uma *socket* de escuta para o atendimento de pedidos de ligação.

```
import select as sel
...
socket_list = [listen_socket]
while True:
      R, W, X = sel.select(socket_list, [], []) # Espera sockets com
      for sckt in R:
            if sckt is listen_socket:     # Se for a socket de escuta...
                  conn_sock, addr = listen_socket.accept()
                  addr, port = conn_sock.getpeername()
                  print('Novo cliente ligado desde %s:%d' % (addr, port))
                  socket_list.append(conn_sock)  # Adiciona ligação à lista
            else:       # Se for a socket de um cliente...
                  msg = sckt.recv(1024)
                  if msg.decode():                    # Se recebeu dados
                        sckt.sendall(msg.decode()[::-1].encode())     # responde
                  else:                         # Se não recebeu dados
                        sckt.close()          # cliente fechou ligação
                        socket_list.remove(sckt)
                        print('Cliente fechou ligação')
...
```

É importante ressaltar que o *select* não serve apenas para monitorizar *sockets*, mas qualquer tipo de objeto que tenha o método `fileno()` (que retorna o *file descriptor* subjacente).

Um cliente que interage com esse servidor não precisa se ligar e desligar em cada interação, como pode ser visto no exemplo abaixo.

```
...
conn_sock = s.socket(s.AF_INET, s.SOCK_STREAM)
conn_sock.connect((HOST, PORT))
while True:
    msg = input('Mensagem: ')
    if msg == 'EXIT':
        break
    else:
        conn_sock.sendall(msg.encode())
        resposta = conn_sock.recv(1024)
        print('Resposta: %s' % resposta.decode())

conn_sock.close()
...
```

Mais detalhes sobre o uso da função *select* podem ser obtidos nos slides da TP03.

2. Processamento de pedidos em paralelo com *socketserver*

A construção de servidores através das classes definidas no módulo *socketserver* pressupõe a definição de uma classe derivada de *BaseRequestHandler*. Entre outros, esta classe permite definir o método *handle* que trata da ligação com o cliente. Relembre o exemplo dado na aula TP04:

```
import socketserver
class MyHandler(socketserver.BaseRequestHandler):
    def handle(self):
        # self.request é a socket ligada ao cliente
        data = self.request.recv(1024)
        print ('ligado a ', self.client_address)
        print (data)
        # Respondemos com a string invertida
        self.request.sendall(data[::-1])
```

Para além do método ***handle***, podem ainda ser definidos os métodos ***setup*** e ***finish***. Sempre que há um cliente ligado, os métodos são chamados pela sequência *setup* → *handle* → *finish*. Estes métodos podem aceder a alguns atributos definidos na classe:

- **self.client_address**: tuplo com o endereço do cliente;
- **self.request**: o objeto socket da ligação que foi aceite pelo servidor (no caso de socket do tipo SOCK_STREAM);
- **self.server**: o objeto de uma classe servidor (TCPServer, UDPServer, ThreadingTCPServer, …).

O trecho de código apresentado acima seria executado após o servidor ter aceitado uma ligação. Como a função *handle* não define nenhum ciclo, a interação entre o cliente e o servidor limitar-se-ia à receção de uma *string* e ao envio da mesma *string* invertida. Após o envio da resposta, a ligação com o cliente seria finalizada.

Para que o cliente pudesse fazer várias sequências de pedido/resposta no contexto da mesma ligação com o servidor, seria suficiente que o servidor incluísse um ciclo na função *handle*, por exemplo incluindo todo o código do método *handle* no âmbito de um **`while True:`**.

---

3. Exercícios fundamentais

1. Copie os programas cliente e servidor apresentados a seguir para sua área e execute-os. Note que estes programas são os mesmos usados no guião anterior, e os alunos que desejarem podem usar os seus projetos ao invés deles.

Cliente (*Cliente.py*)

```
import sys, socket as s

if len(sys.argv) > 1:
      HOST = sys.argv[1]
      PORT = int(sys.argv[2])
else:
      HOST = '127.0.0.1'
      PORT = 9999

while True:
      msg = input('Mensagem: ');
      if msg == 'EXIT':
            break

      sock = s.socket(s.AF_INET, s.SOCK_STREAM)
      sock.connect((HOST, PORT))

      sock.sendall(msg.encode())
      resposta = sock.recv(1024)

      print('Recebi: %s' % resposta.decode())
      sock.close()
```

Servidor (*Servidor.py*)

```
import sys, socket as s

HOST = 'localhost'
if len(sys.argv) > 1:
      PORT = int(sys.argv[1])
else:
      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)

lista = []

while True:
      try:
            (conn_sock, addr) = sock.accept()
            msg = conn_sock.recv(1024)
            resp = 'Ack'

            if msg.decode() == 'LIST':
                  resp = str(lista)
            elif msg.decode() == 'CLEAR':
                  lista = []
                  resp = "Lista apagada"
            else:
                  lista.append(msg.decode())

            conn_sock.sendall(resp.encode())

            print ('lista= %s' % lista)
            conn_sock.close()
      except KeyboardInterrupt:
            break
      except:
            print(sys.exc_info())
            conn_sock.close()

sock.close()
```

[Módulo select]

2. Modifique o programa cliente para que o mesmo interaja com o servidor mantendo as ligações abertas.

3. Modifique o programa servidor para que o mesmo use a função *select* para manter ligações abertas com múltiplos clientes.

4. Modifique o programa servidor para que seja possível entrar com a *string* 'EXIT' e terminar o programa de forma limpa (i.e., sem o uso de Ctrl+C). Dica: monitorize *sys.stdin* através do *select*, juntamente com a *socket* de escuta e as *sockets* de ligação com os clientes.

[Módulo SocketServer]

5. Copie novamente os programas cliente e servidor apresentados no Exercício 1 para a sua área.

6. Modifique o programa servidor através da utilização das classes *BaseRequestHandler* e *TCPServer* do módulo *socketserver*, mantendo todas as suas características.

7. Modifique novamente o programa cliente para que o mesmo interaja com o servidor mantendo as ligações abertas.

8. Use uma classe apropriada do módulo *socketserver* (p.ex., *ThreadingTCPServer* ou *ForkingTCPServer*) para que o servidor possa manter ligações abertas simultaneamente com múltiplos clientes. Note que neste caso a estrutura de dados (a lista) vai ser manipulada concorrentemente pelos vários clientes.

9. Modifique o programa servidor para que seja possível entrar com a *string* 'EXIT', fazendo com que o servidor não atenda mais pedidos de ligação (ver método *shutdown* das classes servidoras do módulo *socketserver*), e que termine assim que os clientes ativos desliguem. Dica: utilize uma *thread* adicional para ler de *sys.stdin*.

---

5. Bibliografia e outro material de apoio

https://docs.python.org/3/library/select.html

https://docs.python.org/3/library/socketserver.html

https://docs.python.org/3/library/threading.html