Practica 2: Sockets y arquitectura cliente-servidor

Miguel Jarne Anguita, Pablo Jiménez Poyatos y María Puyol Romera

7 de abril de 2024

1. Introducción

Para realizar esta práctica, hemos diseñado una arquitectura cliente-servidor usando el paquete sockets de Python. El objetivo de la práctica era permitir que diferentes clientes solicitaran al servidor que realizara ciertas acciones sobre una base de datos sin que hubiera problemas de concurrencia. Para la entrega, hemos creado tres archivos .py. El primero es el archivo del servidor, llamado *servidor.py*; el segundo es la parte correspondiente a los clientes, denominado *cliente.py*; y finalmente, para la creación de la base de datos, hemos utilizado *base_de_datos.py*.

2. Desarrollo

Para realizar el **apartado 3**, hemos creado en nuestro archivo *servidor.py* una función main:

```
def main(srvr_socket: socket) -> None:
    admin = Thread(target=administrador, args= ("",))
    admin.start()
    hebras = []
    while admin.is_alive():
        try:
            # Espera 10s una nueva conexión, sino, vuelve a empezar el bucle.
            cl_socket,_ = srvr_socket.accept()
            hebra = Thread(target=tarea_cliente, args=(cl_socket,))
            hebra.start()
            hebras.append(hebra)
        except:
            pass
    for hebra in hebras:
        hebra.join()
    srvr_socket.close()
```

Programa 1: `servidor.py` (lineas 63–78)

Lo que hago es crear un bucle "infinito" (luego detallaremos más en profundidad el porqué de la creación de una hebra admin y la condición del bucle while) que lo que hace es aceptar nuevas conexiones de los clientes al servidor y crear una hebra que se encarga de atender a cada cliente (después explicaremos porque ponemos try and except). Luego cada hebra se almacena en una lista, la cual la utilizaremos más adelante.

Para realizar el **apartado 4 y 5**, creamos una base de datos (*base_de_datos.py*) con las funciones de dar de alta, dar de baja, consultar o modificar un atributo, guardarla en un archivo o cargar una ya creada. Además, nuestra base de datos es un diccionario clave-valor donde el valor es una lista de 3 atributos + un candado. Estos tres atributos tienen el nombre de atributo1, atributo2 y atributo3.

Para modificar o consultar un registro, el nombre del atributo que se debe introducir por pantalla es el de atributo1, atributo2 o atributo3, sino no, no lo reconocerá y no se consultará nada.

Todas las funciones de la base de datos devuelven una cadena de caracteres, la cual la recibirá el servidor que se la mandará al cliente para mostrarsela al usuario por pantalla.

Para controlar la concurrencia hemos modificado un poco las funciones de la base de datos para que en vez de que se trate toda la base como un solo recurso, se haga bloqueos solamente por registros (por lineas). Veamos en detalle y paso a paso como lo hemos hecho:

```python
from threading import Lock
import csv

class BaseDeDatos:
    def __init__(self):
        self.datos = {}
```

Programa 2: `base_de_datos.py` (lineas 1–6)

Para empezar, importamos del modulo threading los locks y además importamos csv para poder guardar y cargar la base de datos. La clase BasedeDatos se inicializa como un diccionario vacío. Veamos ahora sus funciones una a una.

ALTA:

```python
    def alta(self, clave: str, atributos: list) -> str:
        candado = Lock()
        atri_con_candado = [candado] + atributos
        if clave not in self.datos:
            self.datos[clave] = atri_con_candado

            # Vamos a comprobar que no han entrado varios clientes a la vez a dar de
            # alta en la base de datos a la misma clave.
            atri1, atri2, atri3 = (self.datos[clave][1], self.datos[clave][2],
                                   self.datos[clave][3])
            candado_actual = self.datos[clave][0]
            if (candado_actual == candado and atri1 == atributos[0] and
                    atri2 == atributos[1] and atri3 == atributos[2]):
                return (f"El registro del cliente con clave {clave} se ha registrado "
                        f"correctamente. ")
            else:
                return (f"Error: La clave {clave} ya existe en la base de datos. "
                        f"Inténtelo de nuevo.")
        else:
            return (f"Error: La clave {clave} ya existe en la base de datos. "
                        f"Inténtelo de nuevo.")
```

Programa 3: `base_de_datos.py` (lineas 8–28)

Para dar de alta un nuevo registro, se necesita su clave y su lista de atributos (3 atributos). Ahora bien, el objetivo de la parte 5 es bloquear tan solo el registro al que se le va a realizar alguna acción. Para ello, cada registro necesita tener un candado propio. Es por eso que lo primero que hacemos en la función alta es crear un nuevo candado que va a ser específico para la clave introducida. Este candado lo añadimos a la lista de atributos y se coloca en la primera posición. Ahora bien, para dar de alta un nuevo registro, tenemos que comprobar primero que no está en la base de datos.

En caso de que no esté, lo añadimos. Ahora, como estamos ante un problema de que puede haber varios clientes accediendo a la misma base de datos, tenemos que tener cuidado de que 2 clientes diferentes no quieran dar de alta a la vez a la misma clave pero con diferentes atributos. Esto podría suceder porque cuando el primer cliente consulta la base de datos y ve que no hay ningún registro con esa clave, el segundo podría llegar justamente a la vez y antes de que el primer cliente lo registre en la base de datos, este segundo cliente comprueba que no hay ninguna clave igual en la base de datos. Entonces, ambos habrían entrado en el condicional y se sobrescribirían los atributos de esa clave. Solo estarían registrados los del segundo cliente, aunque a ambos les llegue el mensaje de que se ha dado de alta correctamente. Para controlar este problema, hemos pensado en añadir unas lineas a continuación de manera que se compruebe que los datos que se han registrado son efectivamente los datos que teníamos al principio. Para ello, añadimos dos asignaciones (una simple y otra múltiple) para que así a todos los clientes que hayan entrado en el condicional, les de tiempo a escribir sus atributos. Esto es así porque el tiempo que se tarda en definir el nuevo elemento del diccionario y en consultar sus atributos es más o menos el mismo. Entonces cuando el primero en entrar esté consultando qué candado hay, ya el resto de clientes habrán pasado por la línea self.datos[clave] = atri_con_candado. Una vez en este punto, comprobamos si coincide los valores iniciales con los actuales(finales). En caso afirmativo, se manda un mensaje de confirmación. En caso negativo, todos menos el último recibirán un mensaje de error.

En caso de que esté en la base de datos, manda un error de que ya existe esa clave en la base de datos.

BAJA:

```
31      def baja(self, clave: str, _) -> str:
32          if clave not in self.datos:
33              return (f"Error: La clave {clave} no existe en la base de datos. "
34                      f"Inténtelo de nuevo.")
35          else:
36              try:
37                  candado = self.datos[clave][0]
38                  candado.acquire()
39                  try:
40                      del self.datos[clave]
41                      return (f"El registro del cliente con clave {clave} se ha "
42                              f"eliminado correctamente. ")
43                  except:
44                      return (f"Error: La clave {clave} no existe en la base de "
45                              f"datos. Inténtelo de nuevo.")
46                  finally:
47                      candado.release()
48
49              except:
50                  return (f"Error: La clave {clave} no existe en la base de datos. "
51                          f"Inténtelo de nuevo.")
```

Programa 4: `base_de_datos.py` (lineas 31–51)

Para dar de baja un nuevo registro, comprobamos si esa clave existe o no en la base de datos. En caso negativo, se manda un mensaje de error. En caso afirmativo, "se intenta" coger el candado que tiene asociado ese registro. Digo "intento" porque puede ser que justamente antes alguien lo haya dado de baja. En caso de no conseguir el candado, se manda un mensaje de error. En caso de conseguirlo, lo coge (o espera a que sea su turno para BLOQUEAR ESE REGISTRO ÚNICAMENTE). Cuando obtenga el candado, intenta eliminar ese registro. Si lo elimina, entonces manda un mensaje de confirmación y suelta su candado. Si no ha podido eliminarlo porque otro se ha adelantado, se va por el except y manda un mensaje de error. Igualmente, suelta el candado gracias al finally.

CONSULTAR:

```
54      def consultar(self, clave: str, atributo: list) -> str:
55          relacion_atrib_index = {"atributo1": 1, "atributo2": 2, "atributo3": 3}
56          if clave in self.datos:
57              atributo_minuscula = atributo[0].lower()  # Se coge el primer elem porque es una li
58              if atributo_minuscula not in relacion_atrib_index:
59                  return (f"Error: El atributo {atributo_minuscula} no existe en "
60                          f"la base de datos. Inténtelo de nuevo.")
61              index = relacion_atrib_index[atributo_minuscula]
62              try:
63                  candado = self.datos[clave][0]
64                  candado.acquire()
65                  try:
66                      consulta = self.datos[clave][index]
67                      return (f"El {atributo_minuscula} para el registro con clave "
68                              f"{clave} es: {consulta}")
69                  except:
70                      return (f"Error: La clave {clave} no existe en la base de "
71                              f"datos. Inténtelo de nuevo.")
72                  finally:
73                      candado.release()
74
75              except:
76                  return (f"Error: La clave {clave} no existe en la base de datos. "
77                          f"Inténtelo de nuevo.")
78          else:
79              return (f"Error: La clave {clave} no existe en la base de datos. "
80                      f"Inténtelo de nuevo.")
```

Programa 5: `base_de_datos.py` (lineas 54–80)

Para consultar en nuestra base de datos, necesitamos saber la clave del registro y el atributo. Para empezar, verificamos si la clave se encuentra en nuestra base. En caso negativo, se manda un mensaje de error. En caso positivo, comprobamos si el nombre del atributo es correcto. Después, vemos a cual de los 3 atributos hace referencia y cogemos el índice. Por último, hacemos el mismo proceso que en la baja, intentamos coger el candado y después intentamos consultar el atributo correspondiente.

MODIFICAR:

```
83      def modificar(self, clave: str, valor: list[str]) -> str:
84          relacion_concept_index = {"atributo1": 1, "atributo2": 2, "atributo3": 3}
85          atributo, valor = valor[0].split("->")[0], valor[0].split("->")[1:]
86
87          # Escribo .split() para quitar los espacios en blanco que pueda haber.
88          atributo_minuscula = atributo.split()[0].lower()
89          if atributo_minuscula not in relacion_concept_index:
90              return (f"Error: El atributo {atributo_minuscula} no existe en la base de "
91                      f"datos. Inténtelo de nuevo.")
92          if clave in self.datos:
93              index = relacion_concept_index[atributo_minuscula]
94              try:
95                  candado = self.datos[clave][0]
96                  candado.acquire()
97                  try:
98                      self.datos[clave][index] = "->".join(valor) # Porque hicimos split(->), por
99                      return (f"Se ha modificado correctamente el registro del cliente con "
100                             f"clave {clave} en el atributo {atributo_minuscula}")
101                 except:
102                     return (f"Error: La clave {clave} no existe en la base de "
103                             f"datos. Inténtelo de nuevo.")
104                 finally:
105                     candado.release()
106
107             except:
108                 return (f"Error: La clave {clave} no existe en la base de datos. "
109                         f"Inténtelo de nuevo.")
110
111         else:
112             return (f"Error: La clave {clave} no existe en la base de datos. "
113                     f"Inténtelo de nuevo.")
```

Programa 6: `base_de_datos.py` (lineas 83–113)

Para modificar en nuestra base de datos, necesitamos saber la clave del registro, el atributo y el nuevo valor. Para empezar, separamos nuestra lista valor en atributo y el valor. Después, comprobamos si el nombre del atributo es correcto. A continuación, verificamos si la clave se encuentra en nuestra base. En caso negativo, se manda un mensaje de error. En caso positivo, hacemos el mismo proceso que en la baja y la consulta, intentamos coger el candado y después intentamos modificar el atributo correspondiente con el nuevo valor.

Para usar las siguientes funciones solo se tiene acceso desde el servidor. En particular, solamente puede utilizarlas el administrador (más tarde hablamos de él).

GUARDAR:

```
1       def alta(self, clave: str, atributos: list) -> str:
2           candado = Lock()
3           atri_con_candado = [candado] + atributos
4           if clave not in self.datos:
5               self.datos[clave] = atri_con_candado
6
7               # Vamos a comprobar que no han entrado varios clientes a la vez a dar de
8               # alta en la base de datos a la misma clave.
9               atri1, atri2, atri3 = (self.datos[clave][1], self.datos[clave][2],
10                                     self.datos[clave][3])
11              candado_actual = self.datos[clave][0]
12              if (candado_actual == candado and atri1 == atributos[0] and
13                      atri2 == atributos[1] and atri3 == atributos[2]):
```

```
14                  return (f"El registro del cliente con clave {clave} se ha registrado "
15                          f"correctamente. ")
16              else:
17                  return (f"Error: La clave {clave} ya existe en la base de datos. "
18                          f"Inténtelo de nuevo.")
19          else:
20              return (f"Error: La clave {clave} ya existe en la base de datos. "
21                          f"Inténtelo de nuevo.")
```

Programa 7: `base_de_datos.py` (lineas 8–30)

Para guardar una base de datos, lo único que necesitamos tener es el nombre que le queremos poner al archivo. En cada linea de nuestro archivo, incluimos un registro. Solo guardamos la clave y los tres atributos. El candado no.

CARGAR:

```
1     def alta(self, clave: str, atributos: list) -> str:
2         candado = Lock()
3         atri_con_candado = [candado] + atributos
4         if clave not in self.datos:
5             self.datos[clave] = atri_con_candado
6
7             # Vamos a comprobar que no han entrado varios clientes a la vez a dar de
8             # alta en la base de datos a la misma clave.
9             atri1, atri2, atri3 = (self.datos[clave][1], self.datos[clave][2],
10                                   self.datos[clave][3])
11            candado_actual = self.datos[clave][0]
12            if (candado_actual == candado and atri1 == atributos[0] and
13                    atri2 == atributos[1] and atri3 == atributos[2]):
14                return (f"El registro del cliente con clave {clave} se ha registrado "
15                        f"correctamente. ")
16            else:
17                return (f"Error: La clave {clave} ya existe en la base de datos. "
18                        f"Inténtelo de nuevo.")
19        else:
20            return (f"Error: La clave {clave} ya existe en la base de datos. "
21                        f"Inténtelo de nuevo.")
```

Programa 8: `base_de_datos.py` (lineas 8–30)

Para cargar una base de datos, lo único que necesitamos tener es el nombre del archivo en el que esta guardada. Cada linea del archivo la procesamos y damos de alta a ese nuevo registro.

Veamos ahora más en detalle el código para los clientes y el servidor. Primero se debe ejecutar el servidor y después los clientes:

2.1. SERVIDOR

Después de inicializar el socket, preguntamos si queremos cargar una base de datos o creamos una nueva.

Se nos aconseja para el bucle de aceptar conexiones de nuevos clientes, poner *while True:*. Para evitar poner un bucle infinito, decidimos crear una hebra especial que solo tiene acceso un administrador a través de una clave (1234 en nuestro caso). Esta hebra administrador empieza a ejecutarse al principio de la función principal y lo que hace es preguntar primero la contraseña, después si desea acabar la ejecución del servidor y por último, si desea guardar la base de datos. Para evitar problemas, se espera a que todos los clientes acaben sus peticiones para cerrarlo. Por esta razón, en vez de un bucle while True, ponemos while admin.is_alive() que significa que hasta que el admin no diga que quiere terminar la ejecución del servidor, la hebra seguirá viva.

Además, para evitar que el servidor se bloquee hasta que recibe una nueva conexión para aceptar, le **configuramos el tiempo máximo de espera para aceptar una nueva conexión a 10 segundos**. Esto nos sirve para que si nuestro servidor se encuentra esperando un nuevo cliente y justo el administrador decide cerrar el servidor, no se quede esperando a una nueva conexión para poder cerrarlo. De este modo, cuando de pide cerrar el servidor, como máximo va a esperar 10 segundos para ver si llega una nueva conexión (por eso usamos el try and except, para que no salte el TimeoutError). En caso contrario, sale del bucle y espera a que terminen el resto de clientes.

2.2. CLIENTE

Una vez que ya esta funcionando el servidor y se ha especificado desde la terminal si se desea cargar una base de datos o crear una nueva, los clientes pueden empezar a hacer sus peticiones.

Después de inicializar el socket, se llama a la función *nuevo_ cliente*. En ella, se pregunta la operación que se quiere hacer y además se obtiene el resto de información necesaria. Una vez se obtiene, se organiza el mensaje que se quiere mandar al servidor. A continuación, se recibe la respuesta y se imprime por pantalla. Se vuelve a iniciar el proceso hasta que se escriba salir.