Programación de Sistemas de Telecomunicación
Práctica 1: Universidades y Mazmorras

GSyC

Octubre de 2020

1. Introducción

En esta práctica se actualizará el servidor utilizando utilizando una lista doblemente enlazada para almacenar las partidas.

Además, se añadirán una serie de opciones en el servidor que permitirán imprimir el listado de las partidas en proceso y finalizadas, el número de partidas en proceso y finalizadas, así como cerrar el servidor con una tecla (cerrando todos los clientes que se encuentran jugando en el servidor).

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None
        self.prev = None
```

2. Lista doblemente enlazada

Sera obligatorio en esta práctica que las partidas se almacenen en el servidor mediante una lista doblemente enlazada. La primera diferencia de esta estructura de datos con respecto a la lista enlazada que se ha visto en clase de teoría, es que los nodos tienen un enlace tanto al siguiente nodo, como al nodo anterior. Por lo tanto, a la definición de la clase `Node` de la que ya disponemos, habría que añadir un atributo nuevo para referenciar al nodo anterior.

El segundo cambio consiste en añadir el atributo `tail` en la inicialización de la clase `LinkedList` que hemos creado en clase de teoría. Puesto que ahora los nodos están enlanzados tanto con el nodo previo como con el nodo siguiente (En la lista enlazada sólo estaban enlazados con el nodo siguiente) este atributo cobra sentido, ya que nos permite de una manera sencilla acceder al último elemento e iterar hasta el primer elemento. En una lista enlazada esta acción sólo podía realizarse mediante recursividad.

Si bien algunas de las funciones de la lista enlazada (imprimir, encontrar) funcionarán sin necesidad de realizar ningún cambio, las funciones para insertar los elementos y borrar los elementos deberán ser modificadas para que el enlazado entre los nodos se realice correctamente y se actualice el atributo `tail` cuando corresponda.

Es importante respectar la estructura de datos, es decir, no se puede asumir que los nodos de la lista enlazada contendrán instancias de la clase `Game`.

Es posible que para la codificación de algunas funciones sea necesario utilizar la implementación de `Cursor` vista en clase. De esta manera, podremos iterar sobre la lista y hacer lo que necesitemos con los nodos de la misma. Cuando utilizamos `Cursor`, sí sabemos que los nodos son instancias de `Game` ya que precisamente los cursores se utilizan para poder iterar sobre la lista enlazada fuera de la propia implementación de la lista enlazada.

3. Lista de partidas

El uso de una lista doblemente enlazada cambia ligeramente el modo de guardar y de acceder a cada una de las partidas. Puesto que ahora las partidas van a ser nodos de una lista enlazada y actualmente en la propia partida no hay ningún atributo que pueda ser utilizado como identificador de la partida, será necesario establecer un mecanismo de identificación. Por suerte, si se han seguido las recomendaciones dadas en la P2 y se recuerda lo que se ha hablado en clase, sabremos que la IP y el puerto de un cliente son únicos. De esta forma, dada una tupla (ip, puerto) es posible encontrar la partida en la que se encuentra dicho cliente.

Dicho lo anterior, la variable entera `id` y la variable `client_game` **pasan a estar obsoletas y deben ser eliminadas**. Cada vez que se quiera buscar la partida, se deberá utilizar la tupla mencionada en el párrafo anterior para obtener la partida en la que

se encuentra dicho cliente y devolverla. Para realizar esta operación es posible que se requiera sobrecargar el operador `__eq__` de la clase `Game` haciendo que dicha función compruebe si la tupla se encuentra en la lista de direcciones que hay almacenadas en la partida. Si no se estuvieran guardando las direcciones de los clientes en `Game` se añadirá dicha información a la clase.

En este punto, nos empezamos a dar cuenta de que tenemos en un `Game` muchas listas con la información de los clientes. A saber: nombre, socket, ip. Es necesario (se puntuará) que en lugar de tres listas diferentes se guarde en el `Game` un diccionario que puede llamarse `client_info` que contendrá las claves `client_names`, `client_sockets` y `client_addresses`. De esta manera la información estará más recogida y el código estará más limpio.

4. Servidor

4.1. Comandos

En el servidor actual cada vez que se pulsa una tecla el programa se cierra. Ésto supone un problema ya que en muchas ocasiones podríamos cerrar el servidor sin querer (con consecuencias desastrosas para nuestros clientes). Además, actualmente cuando el servidor se cierra las conexiones de los clientes finalizan inesperadamente. Por otro lado, desde el servidor no podemos hacer realmente nada, ya que no disponemos de ninguna opción que nos permita conocer el estado actual de lo que sucede en nuestra aplicación.

En esta práctica se permitirá que se introduzcan una serie de comandos en el servidor para obtener información además de cerrar el propio servidor de manera consciente. Estos mensajes se imprimen en el propio servidor.

- Si en el servidor se escribe `shutdown` el servidor se cerrará.
- Si en el servidor se escribe `ngames` el servidor mostrará el número de partidas en curso y el número de partidas finalizadas:
- Si en el servidor se escribe `gamesinfo` el servidor mostrará la información de las partidas en curso, en orden inverso a la creación (Primero aparece la partida creada más tarde).

4.2. Cerrar servidor

Si cerramos el servidor de manera abrupta es posible que en el cliente obtengamos la excepción `ConnectionResetError`. Si bien controlando esa excepción solucionaríamos el problema, estaríamos perdiendo la posibilidad de informar al cliente de por qué el servidor ha dejado de responder. La conexión ha podido fallar porque directamente la máquina se a roto o porque nosotros voluntariamente hemos cerrado el servidor para realizar labores de mantenimiento. Es este segundo caso el que queremos contemplar en esta práctica.

Cuando el servidor se cierre voluntariamente utilizando el comando `close` se debe enviar un mensaje de desconexión a todos los clientes para informarles de que el servidor ha sido cerrado por el administrador y que se les desconecta. Para ello, no es necesario crear ningún protocolo nuevo, ya que se puede utilizar el mensaje `Send_DC_Server` creado en la práctica anterior. En este caso no es necesario que el cliente responda con un `Send_DC_Me` ya que el servidor va a apagarse por completo y por lo tanto los `threads` se cerrarán también.

4.3. Ejemplos

```
$ # se escribe  ngames en el servidor:
Active games: 5
Finished games: 2

#se escribe gamesinfo en el servidor:

------ GAME ------
Total Players: 3
Dead Players: 1
Current Stage: 3
Total Stages: 5
--------------------

------ GAME ------
Total Players: 1
```

```
Dead Players: 0
Current Stage: 2
Total Stages: 2
--------------------

------ GAME ------
Total Players: 1
Dead Players: 0
Current Stage: 2
Total Stages: 2
--------------------

#se escribe shutdown en el servidor.
The server has been closed by the admin.

#se recibe un Send_DC_Server en el cliente.
The server has been shut down by the admin. You have been disconnected.
```

5. Condiciones obligatorias de funcionamiento

1. El programa debe cumplir las especificaciones de la P2 (Argumentos, excepciones ....) y añadir la funcionalidad especificada en este enunciado.
2. Los programas deberán escribirse teniendo en cuenta las consideraciones sobre legibilidad y reutilización del código que hemos comentado en clase.
3. Los programas deberán ser robustos, comportándose de manera adecuada cuando no se arranquen con los parámetros adecuados en línea de comandos.
4. Está prohibido asumir que la lista doblemente enlazada contenga instancias de la clase `Game`. El uso de métodos propios de `Game` dentro de la implementación de la clase `DoubleLinkedList` conllevará un suspenso automático.
5. Es **obligatorio** controlar todas las excepciones que puedan surgir durante el transcurso del programa: argumentos inválidos, selección de opciones incorrectas, ctrl + c cuando el programa se está ejecutando, etc.
6. Debe seguirse el modelo de salida mostrado en la sección 4 de este enunciado.

6. Entrega

La práctica puede realizarse en grupos de hasta 3 personas. Sólo hará falta entregar la práctica una vez. Deben aparecer los nombres y apellidos de todos los autores en el fichero main.py en forma de comentario al principio del fichero. Se podrán subir los ficheros de uno en uno, por lo que no es necesario comprimirlos.

El límite para la entrega de esta práctica es el **Jueves, 4 de Febrero a las 23:59** a través de la tarea correspondiente en el AulaVirtual.

La **Prueba de Laboratorio 3** correspondiente a esta práctica se realizará en el aula de prácticas habitual el **Viernes 5 de Febrero a las 11:00**