Ejercicio de clase de TCP. Diccionario.

Se debe entregar un fichero de tipo zip con un proyecto de NetBeans para cada pregunta, situado en un directorio sea el número de la pregunta.

```
ejercicios_clase_TCP_1_apellido2_apellido1_nombre
+--1
+--2
+ (...)
```

Todo el mundo debería hacer los dos primeros ejercicios y entregarlos al final de la clase. Se proponen más ejercicios para quien le sobre tiempo, o para entregarlos en una segunda entrega que se pondrá más adelante.

1. Servidor de diccionario en red.

Se quiere implementar el servidor de un protocolo para un diccionario en red que se puede consultar y modificar. Inicialmente será un servidor de un único hilo. Crear el proyecto de manera que con los valores de parámetros para línea de comandos acepte peticiones por el puerto 7890. No hay que hacerlo en el programa, solo en las opciones del proyecto en NetBeans.

El servidor almacena las palabras y sus significados en un `HashTable`. Como clave se almacena la palabra, y como valor su significado.

Cuando se conecta un cliente, puede enviar comandos al servidor. Cada comando está en una línea de texto. Cada comando recibe una respuesta del servidor en una línea de texto. Los comandos son los siguientes:

Comando	Respuesta	Comentarios
`?`*`palabra`*	`palabra:significado`	Consulta el significado de la palabra.
`!`*`palabra=significado`*	`OK palabra=significado`	Asigna un significado a una palabra. Si la palabra ya está, la nueva definición sustituye a la anterior.
`exit`	`bye`	El servidor cierra la conexión tras recibir este comando y enviar esta respuesta.
`quit`	`bye`	El servidor cierra la conexión tras recibir este comando y enviar esta respuesta.
`otro_comando`	`ERR: otro_comando`	

Se envían todos los comandos al servidor. El cliente lo único que hace es leer comandos, enviarlos al servidor, y mostrar la respuesta recibida.

Utilizar *matching* de expresiones regulares para detectar los distintos comandos y extraer la información relevante de cada uno. A no ser que en el comando no haya ninguna parte variable, en cuyo caso se puede comparar el comando con la cadena correspondiente, sin más.

Hay que desarrollar el programa cliente y el programa servidor.

El programa cliente debe mostrar un *prompt* `dic>` para recibir cada comando.

2. Convertir el anterior servidor en un servidor multihilo. Probarlo con dos conexiones de cliente distintas.

3. Comandos para buscar palabras que empiecen o que terminen con una cadena determinada.

Comando	Respuesta	Comentarios
`?>`*`comienzo`*	Lista de palabras que empiezan con el comienzo dado.	Cada definición en una línea, con el mismo formato que cuando se consulta el significado de una palabra: *`palabra`*`:`*`significado`* La lista termina con una línea vacía. IMPORTANTE: no se admiten *`comienzo`* ni *`final`* vacíos. Los comandos `?>` y `?<`, que en buena lógica podrían devolver la lista entera de palabras, no se admiten.
`?<`*`final`*	Lista de palabras que empiezan con el final dado.	

Consejo: no hace falta hacer cambios importantes ni en el cliente ni en el servidor. La única diferencia es que ahora el servidor envía, como respuesta, una secuencia de líneas, y para terminar la respuesta una línea vacía. Hay que cambiar el cliente para que, en vez de una línea, lea siempre una secuencia de líneas, hasta que lea una vacía. La implementación de los comandos anteriores debe cambiarse en el servidor, para que devuelva después una línea vacía. A las respuestas que contienen un mensaje de error también se le añaden una línea vacía al final.

4. Persistencia rudimentaria en el servidor. Cada vez que se modifica algo en el servidor, se graban los contenidos del `HashMap` en un fichero de texto con nombre `diccionario.txt`. Utilizar la clase `FileWriter`. Se puede construir un `BufferedWriter` sobre el `FileWriter` para escribir línea a línea. Las líneas deben tener el mismo formato que las respuestas a la consulta del significado de una palabra, es decir:

`palabra``:`*`significado`*

Cuando arranca el servidor, carga las definiciones de este fichero.

Se añade un comando `delete`, que borra todas las definiciones del `HashMap`. Después, como es lógico, hace que el fichero quede vacío. Para ello debería servir el mismo código de programa del que ya se dispone para grabar en el fichero los contenidos del `HashMap`, una vez que este ha quedado vacío.

5. Comando `get_all`. El servidor responde con todas las definiciones. Justo lo que, en buena lógica, haría el comando `?>` si se admitiera.

6. El servidor no cambia. Pero ahora en el cliente se puede especificar, opcionalmente, un nombre de fichero para el comando `get_all`. Es decir: `get_all` *`nombre_fichero`*. Si se especifica el nombre de fichero, el servidor envía el comando `get_all` al servidor. Pero en lugar de enviar la respuesta del servidor a la salida estándar, la envía a un fichero con ese nombre, que crea. Si el fichero ya existe, lo sobrescribe. No hay que hacer nada especial, esta es la funcionalidad por defecto de la clase `FileWriter`, a no ser que se le pase un parámetro determinado al constructor.

7. El servidor tiene un comando para cargar un fichero con definiciones. Este fichero tiene el mismo formato que el que se obtiene con el comando `get_all`. El comando que se introduce en el cliente para ello es `upload` *`nombre_fichero`*. El cliente da un error si el fichero no existe. El cliente no verifica los contenidos del fichero. Sencillamente hace lo siguiente:

a) Envía una línea de texto al servidor con el formato `upload` *`num_bytes`*, donde *`num_bytes`* es el número de bytes del fichero (utilizar la clase `File` para obtener esta información).

b) Envia el contenido del fichero, byte a byte, por el `OutputStream` asociado al `Socket`.

El servidor, en cambio, lee lo que le envía el cliente línea a línea. Una vez leída una línea, puede obtener el número de bytes de que consta utilizando métodos de la clase `String`. De esa manera, puede leer texto línea

a línea pero llevar el control de los bytes leídos para dejar de leer líneas cuando alcanza el número de bytes que el cliente le ha comunicado previamente, en la línea que le ha enviado antes del fichero. Si sucede cualquier excepción durante el proceso, envía una línea `ERR: texto_error`, donde el texto de error es un texto que debe obtenerse con el método `getMessage()` de la clase Exception. Con la primera línea incorrecta que encuentre, debe enviar una línea `ERR: línea num_linea incorrecta: texto_linea`, donde *`num_linea`* es el número de línea y *`texto_linea`* el texto que hay en la línea. Si todo va bien, devuelve sencillamente una línea con el texto OK, después de cargar todas las definiciones. Estas se añaden, una a una, al diccionario, bien creando una nueva entrada o bien modificando el significado para una entrada existente. Es decir, el servidor procesa todo el fichero previamente. Si todo está bien, carga sus contenidos en el `HashTable`. Si cualquier cosa está mal o sucede cualquier excepción, deja el `Hashtable` como estaba. Y en cualquier caso, la repuesta que recibe el cliente es una única línea de texto.

Indicaciones para la prueba

Se puede utilizar el comando `nc`. De hecho, se recomienda probar el servidor con él, antes de desarrollar el cliente. Como parámetro se le pasan el host y el puerto.

Se puede enviar una secuencia de comandos al servidor. Esto permite automatizar las pruebas. Por ejemplo, se puede crear un fichero de texto `comandos.txt` con el siguiente contenido.

```
silla:mueble para asentarse
comando incorrecto
?silla
?mesa
silla:mueble para sentarse
?silla
get_all
```

Estos comandos se pueden enviar al servidor directamente con:

```
nc host puerto < comandos.txt
```

Y `nc` mostraría las respuestas recibidas del servidor.