Tarea 3.1

Motor de base de datos no relacional en red

Plazo límite: martes 5 de noviembre a las 23:59 hrs., por Git.

Repositorio disponible en `https://classroom.github.com/g/r34HcEJY`.

1. Objetivo

Esta tarea podríamos considerarla una continuación en espiral de la tarea 1 del curso. Nuestro objetivo es desarrollar una base de datos no relacional, con arquitectura cliente-servidor, permitiendo que múltiples clientes concurrentes conectados por red (internet) puedan conectarse al servidor de base de datos para realizar operaciones de lectura y escritura de datos. La primera parte de esta tarea será enteramente teórica y consistirá en diseñar con formalidad el protocolo de capa aplicación en que se basará la comunicación cliente-servidor. La segunda parte de la tarea consistirá en implementar el protocolo mediante aplicaciones cliente y servidor, a desarrollar con un lenguaje de más alto nivel (Python 3) que en la tarea 1.

2. Descripción

En esta tarea deberás diseñar y luego implementar (en la parte 2) un protocolo básico para comunicar la aplicación cliente y el servidor de la base de datos K-V. En la tarea sólo se requiere diseñar el protocolo, no las aplicaciones mismas. Las restricciones básicas para diseñar el protocolo son las siguientes:

- Debe funcionar en la Internet pública, y permitir que usuarios en distintas redes IP puedan conectarse a un servidor.
- El servidor debe permitir conexión desde múltiples clientes.
- El protocolo debe definir claramente los tipos de mensaje, formatos de mensaje, estados y acciones posibles. Se debe considerar que el protocolo opera estrictamente en capa aplicación, por lo que se sugiere utilizar formatos de mensaje inspirados (no una mera copia) en otros protocolos como HTTP.

2.1. Requisitos

La primera parte de la tarea consiste en diseñar el protocolo de red de acuerdo al contexto antes descrito. El entregable debe consistir en un informe que defina los siguientes aspectos:

1. Tipos y formatos detallados de mensaje intercambiados en las interacciones entre los hosts contemplados en el protocolo. Los formatos de mensaje pueden combinar distintos formatos de representación, por ejemplo, texto y datos binarios. Se deben definir los tipos de mensaje considerando los distintos campos posibles. A su vez, esto requiere definir:

2. Interacciones posibles en el protocolo. De preferencia, ilustrar con diagramas formales, por ejemplo, UML de secuencia y/o de estado.

La definición del protocolo no debe realizar supuestos detallados sobre la arquitectura de software que permita implementarlo. Por ejemplo, no debe entrarse en detalles sobre la interfaz de usuario o lógica de aplicación en los extremos. El foco debe estar en los tres aspectos mencionados arriba.

Una vez entregado el informe de la primera parte de la tarea, cada grupo recibirá su informe en formato electrónico con comentarios y posiblemente una minuta con recomendaciones. Además, los grupos podrán reunirse con el profesor y/o el ayudante para resolver dudas y mejorar su especificación. Es importante que los grupos antes de implementar el protocolo en la segunda parte de la tarea tengan claridad sobre el diseño del protocolo.

En la segunda parte de la tarea, el foco estará en el desarrollo de una versión funcional del protocolo propuesto. Estará permitido realizar la implementación en lenguaje Python 3.

3. Evaluación

La evaluación de la primera parte de la tarea se realizará mediante rúbrica que contemplará los siguientes criterios:

30 % Definición de tipos de mensaje en el protocolo.

40 % Definición de acciones del protocolo.

30 % Definición de estados del protocolo, incluyendo el manejo de situaciones de excepción.

Cada uno de estos ítems será evaluado en una escala 1-10.

3.1. Modalidad de Trabajo

La tarea debe ser desarrollada en grupos de dos (parejas). Cada integrante debe contar con una cuenta de usuario en GitHub. El desarrollo debe realizarse por ambos integrantes, y deben quedar claramente registradas sus operaciones de *commit* en el sistema, con comentarios descriptivos en cada una de estas operaciones.

Para la primera parte de la tarea, bastará la entrega del informe en formato PDF en el directorio raíz del repositorio. El archivo debe llamarse "informe_parte1.pdf".

Para obtener un repositorio con el código base, los integrantes de cada grupo deben acceder a: `https://classroom.github.com/g/r34HcEJY`.

4. Anexo: Requisitos Funcionales de Cliente y Servidor

Se entregan los requisitos funcionales de aplicaciones cliente y servidor descritas en la tarea 1, a fin de alinear el diseño del protocolo con dichos requisitos.

4.1. Generación y manejo de claves

1. La clave al insertar un valor puede ser autogenerada por la base de datos, o entregada por el usuario. En caso que sea entregada por el usuario siempre debe ser un valor entero positivo.

2. Si la clave es autogenerada, se usa un contador global en la base de datos para ello. Al iniciarse la base de datos, el contador debe comenzar con valor entero aleatorio positivo entre 1000 y 10000. El contador global se incrementa después de generar una clave.

4.2. Proceso Servidor

1. El proceso del servidor es iniciado desde la línea de comandos. Se puede entregar como argumento (`-s`) la ruta al socket de dominio a utilizar. Si no se especifica el argumento (`-s`), se debe usar un socket en la ruta `/tmp/db.tuples.sock`.

2. El servidor escucha una conexión de un cliente. Cuando un cliente se conecta, acepta la conexión y luego queda esperando comandos del cliente. ~~El servidor en esta versión de la tarea sólo puede atender un cliente a la vez.~~

3. El servidor es capaz de recibir por el socket un solo comando, procesarlo, y retornar al cliente el resultado de la operación.

4. El cliente puede desconectarse del servidor. Cuando esto ocurre, el servidor debe continuar esperando la siguiente conexión de un cliente.

4.3. Proceso Cliente

1. Al igual que el proceso del servidor, el proceso cliente es iniciado desde la línea de comandos. Se puede entregar como argumento (`-s`) la ruta al socket de dominio a utilizar. Si no se especifica el argumento (`-s`), se debe usar un socket en la ruta `/tmp/db.tuples.sock`.

2. El cliente al iniciarse tiene un tiempo máximo (tiempo de *timeout*) de 10 segundos para conectarse al servidor. Si dicho tiempo expira, se debe mostrar un error al usuario y esperar el siguiente comando.

3. Los comandos que se pueden ejecutar desde el cliente son los siguientes:

   - `connect`: Requiere la ruta al socket mediante el cual se conectará el cliente al servidor.

- `disconnect`: Permite al cliente desconectarse del servidor.
- `quit`: Cierra la conexión actual (si la hay), y cierra el cliente.
- `insert(key,value)`: Inserta una nueva tupla clave-valor. Se genera error si la llave ya existe en la base de datos.
- `insert(value)`: Inserta `value` en la base de datos, creando un nuevo par clave-valor con clave autogenerada. Retorna la clave generada si la operación tiene éxito.
- `get(key)`: Retorna el valor asociado a la clave `key`.
- `peek(key)`: Retorna '`true`' si la clave está registrada en la base de datos, o '`false`' si no lo está.
- `update(key,value)`: Permite actualizar el valor asociado a la clave `key`. Retorna error si la clave ya está en la base de datos.
- `delete(key)`: Permite eliminar un par clave-valor con clave `key`. Debe retornar valor si la clave no existe.
- `list`: Retorna un listado con todas las claves disponibles.

4.4. Conexiones concurrentes de clientes al servidor

Para implementar conexiones concurrentes de clientes al servidor, éste último debe ser aumentado considerando las siguientes características:

- Aceptar múltiples conexiones de clientes: El servidor debe correr en su thread principal un ciclo que permita escuchar conexiones de múltiples clientes a través de un socket IP. Al aceptar una conexión, se desencadenan las siguientes acciones en el servidor:
  1. el servidor crea un client socket, desde el cual le es posible leer y escribir datos desde/hacia un cliente conectado.
  2. Luego el servidor crea un thread que procesa los comandos de la sesión cliente a través del client socket generado.
  3. El thread debe correr un loop que espere un comando desde el cliente, lo ejecute y retorne el resultado por el socket. Es el mismo comportamiento de la tarea anterior con el cliente, pero ahora en threads y clientes paralelos.
  4. Cuando el cliente ejecuta `disconnect`, se debe cerrar el client socket y además el thread que procesa su sesión en el servidor debe terminar.
- Sincronización de las estructuras de datos. Se trata esto en mayor detalle en la siguiente sección.

4.5. Sincronización de contador y de estructuras de datos

Dado que habrá múltiples clientes conectados al servidor es necesario sincronizar el acceso a la base de datos, a fin de evitar anomalías causadas por condiciones de carrera en las secciones críticas del código del servidor.

El contador de claves autogeneradas debe ser sincronizado como un contador exacto. Por otro lado, el almacenamiento de pares clave-valor debe ser sincronizado minimizando el sacrificio de rendimiento. Para esto, lo razonable es utilizar locks de lectura y escritura, los cuales están disponibles en la API POSIX. En lenguaje Python, existe el paquete `readerwriterlock` que permite utilizar locks de lectura y escritura.