Redes de Computadoras

Curso 2022

Grupo 16

Informe - Obligatorio 2

Autores:
Alexis Baladon
Miguel Machado
Mathias Martinez

17 de octubre de 2022

Índice

1. Introducción

En este informe se presentan el diseño y la arquitectura de un problema en el cual diversos clientes utilizan los servicios de una base de datos distribuida, que está compuesta de varios servidores pertenecientes a una misma red local. Esta base de datos tiene como objetivo almacenar y/o entregar datos cuyo formato es <**clave, valor**>. La idea es que cada cliente se comunique con la base de datos a través de uno de los servidores, especificando la dirección ip, y un puerto específico para el cual dicho servidor aceptará mensajes de los clientes. Cuando un servidor reciba este tipo de mensajes los procesará, identificará qué tipo de solicitud le están haciendo, y en base a ello podrá efectuar una operación sobre los datos que le soliciten, o un mensaje de error en caso de no haber comprendido la solicitud.

Luego, entre los pares de servidores que conforman esta base, se distribuirán la información con el objetivo de repartir su carga. En principio, los servidores podrán unirse a la red en cualquier instante, ya sea porque se trata de un servidor nuevo el cual es agregado por primera vez, o un servidor ya existente el cual perdió la conexión e intenta volver a conectarse. Esto quiere decir entonces que se necesitará disponer de una técnica para que los servidores existentes en la red detecten cuando un nuevo servidor aparece, conocer sus datos para conectarse a él, y finalmente establecer una conexión para poder intercambiar datos. A esta forma en la que nuevos servidores son descubiertos se le llama DESCUBRIMIENTO, y será uno de los protocolos a estudiar en las próximas secciónes.

Como fue mencionado, cuando un cliente se conecta a la base distribuida lo hace a través de un servidor en específico, sin importar si es este último el propietario. Es así que, cuando es a otro servidor al que le corresponde procesar la solicitud surge la necesidad de que el primero se comunique con el servidor designado mediante el protocolo DATOS del mismo modo que haría un cliente.

En lo que sigue se expone una breve descripción del diseño del problema y la implementación realizada en python, que ofrece una librería para la utilización de la API de sockets TCP y UDP [2].

2. Modelado de una base de datos distribuida en una red local

Se desea diseñar una aquitectura que siga el siguiente esquema:

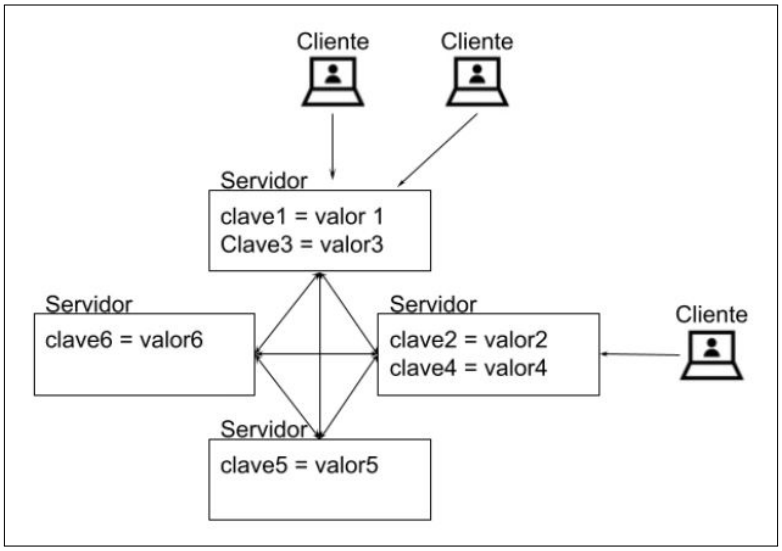

Figura 1: Ejemplo de base de datos distribuida, que almacena datos en la forma <clave, valor>.

Además se desea utilizar dos protocolos: uno para que entre un cliente y servidor (o entre pares) puedan intercambiar mensajes para las operaciones (DATOS), y el otro para que un nuevo par pueda anunciar su aparición en la red (DESCUBRIMIENTO).

2.1. Protocolo DATOS

Este protocolo se encarga de que tanto emisor como receptor acuerden un formato de mensaje para manipular datos de la red de pares. Los socket que utiliza para la comunicación obedecen al protocolo TCP de la capa de transporte. Esto quiere decir que, cuando un cliente inicia la conexión con el servidor, este primero acepta la conexión y luego reserva un socket de uso exclusivo, para el intercambio de mensajes de operaciones con ese cliente. La figura 2 muestra el *socket de acogida* el cual está destinado a recibir las distintas solicitudes de conexión, tanto de un

cliente como de un peer. De aceptarse correctamente cada una de estas conexiones, el servidor reserva un nuevo socket de uso exclusivo para el intercambio de información con cada host conectado a él.

El número de puerto con el que el servidor define este socket de acogida , lo exhibe a los demás servidores como parte del mensaje del protocolo que se menciona en la siguiente subsección.

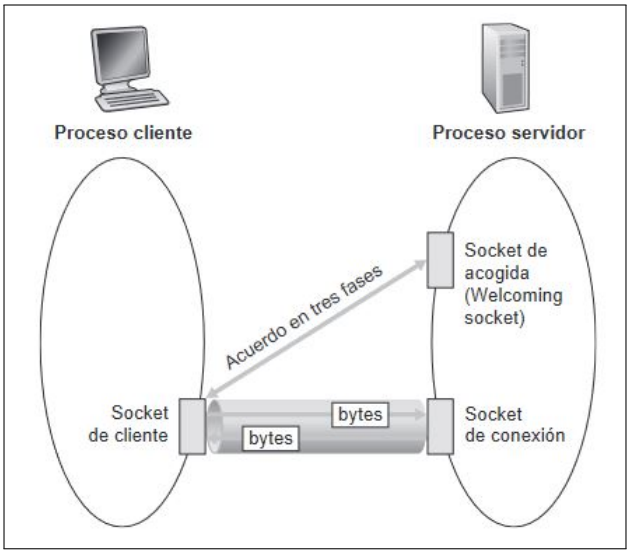

Figura 2: Procedimiento de establecimiento de una conexión TCP para soportar datos.

2.1.1. Especificación del protocolo

A continuación se especifica la sintaxis y semántica del protocolo, al igual que respuestas esperadas para cada mensaje:

```
GET <key>\n
```

Dada una clave ***key*** devuelve un mensaje de confirmación ***OK*** <***value***>***n***, donde ***value*** es el valor asociado a la clave. Si no existe esta clave en la base, retorna el mensaje ***NO\n***.

```
SET <key> <value>\n
```

Agrega el dato <***key, value***> en la base si es nuevo, o modifica uno existente que como clave tenga ***key***. Devuelve un mensaje de confirmación ***OK\n***.

```
DEL <key>\n
```

Elimina la entrada con clave ***key*** de la base. Devuelve un mensaje de confirmación ***OK\n***. Si no existe esta clave en la base, retorna el mensaje ***NO\n***.

2.2. Protocolo DESCUBRIMIENTO

A diferencia del protocolo anterior, este es de uso único por parte de los servidores. Cuando un nuevo servidor se suma a la red local, emite cada cierto lapso de tiempo (p.ej. 30 segundos) un único mensaje hacia todos los servidores de esta red, por un puerto especificado. Es decir, se anuncia hacia los servidores ya existentes a través de un mensaje de broadcast.

Los sockets que este protocolo utiliza son UDP (sobre estos sockets se generan los mensajes de BROADCAST). Uno de estos sockets se encargará de anunciar la existencia del servidor mediante mensajes ANNOUNCE, mientras que otro recibirá los anuncios de otros pares con el fin de controlar cuáles están activos en la red, determinar de qué servidores mantener información, y si abrir o cerrar conexiones TCP para soportar el protocolo DATOS.

2.2.1. Mensajes de este protocolo

Tal y como se acaba de mencionar, un servidor se anuncia por medio de un único mensaje. Dicho mensaje debe cumplir el siguiente formato:

```
ANNOUNCE <port>\n
```

en donde <***port***> es el puerto por el cual el servidor acepta conexiones para el protocolo DATOS.

Se establece un puerto por defecto por el cual se emite este mensaje en caso de que se omita su especificación desde la ejecución por comando (por ejemplo, un puerto por defecto puede ser 2022). A los efectos prácticos, en la implementación este puerto está definido como una constante dentro del código que implementa a DESCUBRIMIENTO, para poder cambiarlo por uno a elección.

Evidentemente, tanto para este protocolo como para el de DATOS es preferible utilizar puertos altos (numeros mayores o iguales que 1024). Los puertos conocidos (entre 0 y 1023) están destinados para los protocolos más populares (80 en HTTP, 22 en SSH, 25 en SMTP, etc), y por ello se evita su uso.

Todos los demás servidores podrán escuchar por este mensaje, y si se recibe correctamente deberán determinar si alguno de sus datos debe ser reasignado al nuevo servidor. Ni bien se determina cuales son estos datos, son envíados al dicho servidor haciendo uso de la operación SET del protocolo DATOS.

Por último, cabe aclarar que el inicio de la conexión y la entrega de los datos al servidor responsable no se hace de forma bidireccional entre los pares anunciante y descubridor. Esto es, el servidor anunciante envía su ANNOUNCE para que los demás peers lo reconozcan, establezcan una nueva conexión, y le entreguen a este los datos que correspondan. Sin embargo, para este poder establecer una conexión hacia ellos (los cuales lo descubrieron), debe esperar los respectivos mensajes de ANNOUNCE de cada uno, siendo él esta vez quien cumpla el rol de descubridor. Como consecuencia, cada servidor en la red será tanto un anunciante como un descubridor de nuevas instancias de servidores.

3. Implementación con sockets, clases implementadas y sus funcionalidades

La implementación del servidor fue realizada mediante una arquitectura en capas según se muestra en la imagen a continuación.

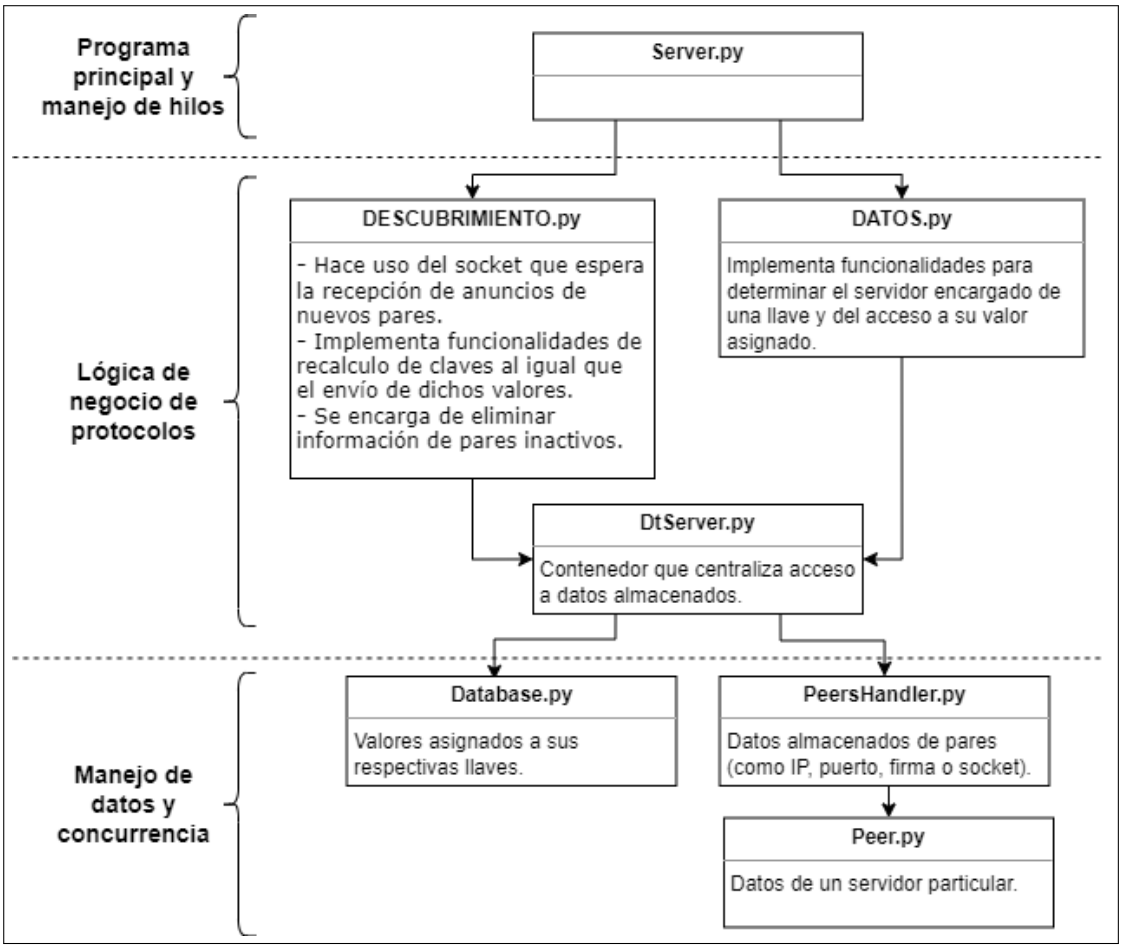

Figura 3: Diagrama de dependencias de servidor simplificado.

3.1. Capa principal y manejo de hilos

Para no entrar tanto en detalle de lo que sería la implementación a bajo nivel, se procede con una explicación breve de las acciones que se realizan en este módulo.

Aquí es donde los hilos juegan un papel muy importante, puesto que es necesario que el servidor realice varias tareas de forma concurrente. Estas tareas son (además del hilo principal del propio servidor): atender o estar constantemente atento a la aparición de nuevas instancias de servidor, notificar de su existencia a los demás servidores de la red cada 30 segundos, aceptar conexiones de clientes y procesar solicitudes de los mismos. Por lo tanto, el programa principal del servidor

ejecuta las siguientes tareas:

- Inicializa el servidor con los argumentos pasados al ejecutar desde la terminal (controlando que la cantidad de argumentos recibida sea válida).
- Instancia sockets UDP para escuchar y envíar mensajes de BROADCAST.
- Crea un hilo destinado al announce, y lo pone en ejecución.
- Crea un hilo destinado al descubrimiento, y lo pone en ejecución.
- Crea un hilo que ejecuta un bucle para aceptar conexiones de clientes, y lo pone en ejecución.
- Imprime en consola su estado: si está conectado, indica los puertos donde escucha los diferentes mensajes.
- Imprime en consola en cada momento cuando detecta la presencia de mensajes, tales como: notificar que detectó un nuevo servidor, o el resultado de una operación en caso de tratarse de un mensaje de datos.

Para poder llevar a cabo estas tareas, el servidor consume los servicios que se mencionan en la siguiente subsección.

El servidor finaliza únicamente cuando se fuerza la detención del programa principal, o en caso de falla (como falta de suministro de energía en el host).

3.2. Lógica de negocio

Esta capa hace uso de funcionalidades de parsing para controlar que las entradas recibidas por otros servidores tengan un formato válido.

En cuanto al protocolo de descubrimiento, se tienen ciertas consideraciones para casos no especificados en la letra de la tarea. En primer lugar, se eliminan pares inactivos para lo que se utiliza una función bloqueante acquire para poder eliminar servidores inactivos y cerrar sockets sin que se esté haciendo uso del mismo. Más allá de este caso particular, bajo el punto de vista de la capa antes mencionada no deben hacerse controles de concurrencia sobre estructuras, ya que se asume que la implementación de la capa inferior ya se encargá de ello. En cualquier caso, con el fin de evitar deadlocks se toma la postura de no realizar un lock a menos que el código dentro no depende de otro hilo o proceso, y en caso de hacerlo, existirán

temporizadores que salgan del blockeo.

Por último, en lo que respecta al servidor que ya no es escuchado en la red local, si pasaron 60 segundos desde que hizo su útlimo ANNOUNCE, cada peer que conocía anteriormente a dicho servidor lo estará eliminando de su lista de pares, y cerrará la conexión del socket que antes había establecido hacia él para soportar datos. Esto solucionaría el problema de conservar todo el tiempo peers en los servidores, de los cuales se desconoce su estado de conexión transcurrida una cantidad importante de tiempo.

3.3. Manejo de datos y concurrencia

Con el fin de no permitir comportamientos indeseados al acceder de forma concurrente a las colecciones almacenadas en memoria se utilizan locks y se crean copias de cada estructura retornada. Dado que la capa superior desconoce la implementación de dichos controles, todo lock adquirido deberá ser liberado en la misma función donde se utilice.

Otra cosa que se tuvo en cuenta con los datos, fue **determinar que hacer con aquellos que pertenecieran a un servidor que se acaba de caer**. En estos casos, la desición tomada es que **las claves directamente se pierden**. El problema que generaba el dejar los datos replicados en más de una base, se logró deducir cuando se probó instanciar tres o más servidores, descubriendo un caso en el que al último servidor anunciado se le entregó una clave con valor desactualizado.

Entonces, cuando un cliente solicitaba el valor por esta clave (sin importar a que servidor), la consulta llegaría a este servidor por ser el designado, por lo que este encontraba el valor y lo hacía llegar al cliente. Sin embargo, el cliente había recibió una versión vieja del valor para la clave dada.

4. Comandos

Tanto clientes como servidores son lanzados a través de un comando con sus respectivos parámetros. A continuación se muestran ejemplos de ejecución en python

(en los siguientes ejemplos se muestra la ejecución con la versión python3).

4.1. Ejecutar un servidor

Para el caso de un servidor, su instanciamiento es producido en base a la invocación del siguiente comando:

```
python3 server.py localhost 3024 2022 2023
```

donde localhost el la dirección ip del equipo local, 3024 es el puerto donde acepta conexiones para el protocolo DATOS, 2022 es el puerto elegido para emitir mensajes de announce al broadcast y 2023 es el puerto que admite para recibir los announce de otras instancias de servidores. Notar que en el ejemplo anterior de invocación se utilizó localhost como dirección para el servidor, pero podría especificarse una dirección ip manualmente si se desea. También se admite la ejecución del comando sin la especificación de los puertos. Por ejemplo, si se realiza la invocación del comando omitiendo los puertos:

```
python3 server.py localhost
```

el servidor comienza la ejecución con unos valores de puertos por defecto, ya sea para datos, announce y discover, hardcodeados en el código que implementa al propio servidor.

En todos los demás casos, si se intenta ejecutar un servidor con demasiados parámetros (más de 4) o con error de formato en alguno de ellos (ip, puertos), la consola devuelve un error.

También ofrece un menú de ayuda de como armar una línea de ejecución válida, por medio de la siguiente invocación:

```
python3 server.py -h
```

4.2. Ejecutar un cliente

Para el cliente se ofrecen dos "sabores"de ejecución por comandos.

La primera toma los argumentos necesarios para hacer la consulta a la base de datos directamente.

```
python3 client.py 127.64.10.8 3024 SET 1234 Hola
```

En el ejemplo de invocación anterior, el cliente elige al servidor con ip 127.64.10.8 y da por hecho que tiene un puerto número 3024 para aceptar su solicitud. Los parámetros restantes correponden a la operación en cuestión. Si no se encuentra al servidor con ip 127.64.10.8 se genera un mensaje indicando que hubo un error durante la conexión. Y si se encuentra la ip pero el puerto no es uno que el servidor admite, se genera un mensaje de error indicando que se le denegó la conexión. Al igual que con el comando de la invocación de un servidor, al momento de la ejecución del comando para el cliente se controla la cantidad de argumentos y el formato de los mismos.

En caso de establecerse la conexión satisfactoriamente, la retroalimentación de la operación (en el ejemplo anterior: SET 1234 Hola) es la anteriormente descrita en la **subsección 2.1.1**.

La segunda forma de invocación del comando ofrece una interfaz de menú donde el usuario que ejecuta al cliente puede ir seleccionando diferentes opciones: tipo de operación, ip y puerto del servidor a consultar, y operación con la clave y/o valor.

```
python3 clientCLI.py
```

Una vez el usuario selecciona los parámetros necesarios, el programa genera el mensaje de DATOS y realiza el envío a la dirección ip y puerto que se hayan establecido. A partir de ahí, el comportamiento de la interacción cliente y servidor es practimante igual al caso anterior.

La única ventaja que se puede rescatar de ejecutar el cliente en esta versión de comando, es que tras finalizar una petición de datos sobre un servidor, el cliente puede seguir activo para realizar otra petición (a menos que el usuario decida cerrar el cliente, opción que también se ofrece en la interfaz).

Referencias

[1] *Kurose, James, y Keith W. Ross. Redes de computadoras. Vol. 7. Pearson Educación, 2017*

[2] *Python Sockets Interface. https://docs.python.org/3/library/socket.html*