UNIVERSITAT POLITÈCNICA DE VALÈNCIA

CAMPUS D'ALCOI

DSIC

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Sistemas de almacenamiento y procesado distribuido

Laboratorio 2

Javier Esparza Peidro – jesparza@dsic.upv.es

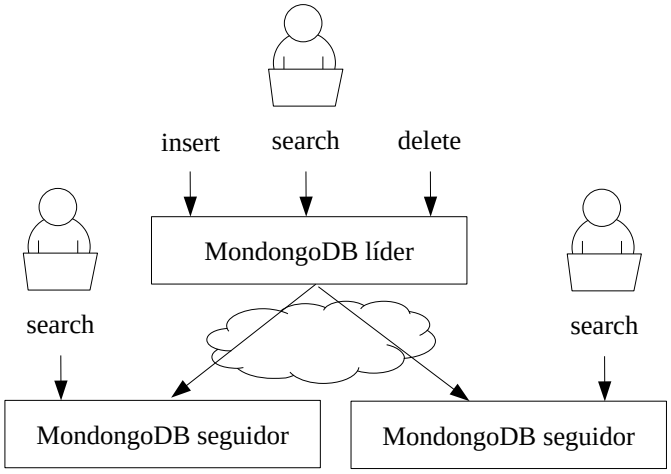
Contenido

1. Introducción

En este laboratorio se continúa con el proyecto iniciado en el laboratorio anterior. El objetivo consiste en extender el almacén de documentos local MondongoDB para convertirlo en un almacén de documentos replicado tolerante a fallos.

En concreto, se cubrirán los siguientes objetivos parciales:
1. El almacén de documentos será accesible a través de la red.
2. Existirán múltiples réplicas que implementarán un esquema de replicación pasiva, con un líder y *n* seguidores.
3. El almacén podrá recuperarse tras suceder un fallo.

La siguiente figura ilustra esta idea.



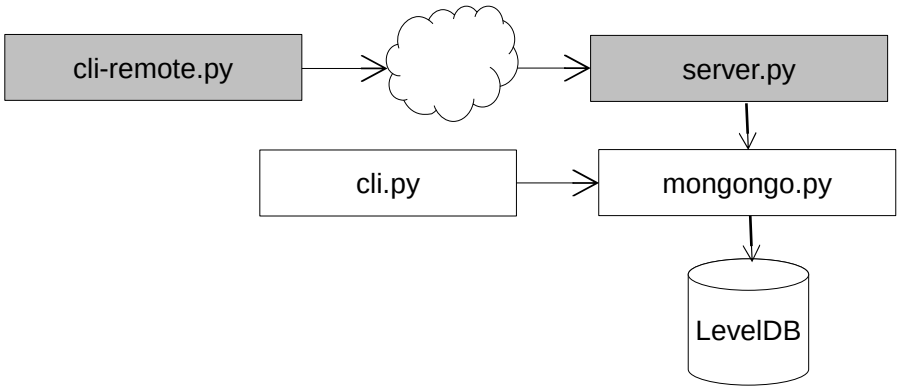
En la sección 2 se plantea una estrategia para convertir el almacén local en un almacén accedido a través de la red. En la sección 3 se implementa un cliente para el almacén. En la sección 4 se propone implementar un almacén replicado por medio de replicación pasiva. En la sección 5 se discute brevemente acerca del protocolo de recuperación y tolerancia a fallos que el almacén debería de implementar.

2. El servidor

En este laboratorio se implementarán los siguientes módulos adicionales:

- *server.py*: contiene el servidor replicado, que después accede al almacén de documentos implementado en el laboratorio anterior. Publica las operaciones del almacén de documentos como una API accesible a través de la red (usando sockets).
- *cli-remote.py*: es un cliente que accede al servidor replicado a través de la red y permite efectuar las operaciones publicadas por éste (las operaciones del almacén de documentos).

A continuación se ilustran las dependencias entre módulos del almacén replicado. En gris se resaltan los módulos que se implementarán en este laboratorio.



En el interior del módulo *server.py*, se sugiere implementar la clase *Server*. Esta clase implementa el servidor remoto. A continuación se presenta una guía para implementar las distintas operaciones de esta clase:

El módulo *server.py* publica la siguiente API:

`server.connect(path, opts:dict={})`
Crea una nueva conexión contra el almacén replicado y devuelve un objeto de la clase *server.Server*. El *path* es un string que determina el path del almacén de documentos. Además, se sugieren las siguientes opciones de configuración:

- *host*: es un string que determina la interfaz de red y el puerto (ej. “localhost:9090”) en el que escuchará peticiones de entrada.
- *leader*: es un string que determina el host en el que escucha el líder (ej. “localhost:9090”)

`class server.Server`
Representa una conexión abierta contra el almacén.

`server.Server.run()`
Este método arranca el servidor, que empieza a escuchar conexiones de entrada. Para las comunicaciones se utilizará el paquete *socket*. Para implementar esta parte se puede utilizar un código parecido al siguiente:

```
import socket
...
self.serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
self.serversocket.bind((self.host, self.port))
self.serversocket.listen()

while True:
  (clientsocket, address) = self.serversocket.accept()
  print("[Server] client connected!")
  self.serve(clientsocket)
```

Como se puede observar, el socket servidor recibe las conexiones cliente una tras otra. El método *serve()* se encarga de servir las peticiones.

`server.Server.serve(sock)`
En este método está el grueso del servidor. Analiza las peticiones de entrada y las sirve. Para ello, debe de leer la petición del socket cliente, ejecutar el comando solicitado, y devolver una respuesta. Se asumirá que tanto las peticiones como las respuestas están codificadas en listas en formato JSON.

En general, las peticiones poseen la siguiente estructura:

[tipo_operación, arg1, arg2 …]
Y las respuestas tendrían la siguiente estructura:

["ok", resultado] ó ["ko", motivo]

Para simplificar, estos mensajes serán codificados en string, utilizando el formato JSON. Podemos hacerlo utilizando la primitiva json.dumps().

Por ejemplo, a continuación se lista una secuencia de petición-respuesta en la que se añade un documento a la colección “users”, y posteriormente se busca:

```
["insert","users",{"email":"a","name":"a","age":18}] → ["ok","1234"]
["search","users",{"email":"a"}]  → ["ok",[{"email":"a","name":"a","age":18}]]
```

Para trabajar con un socket, se recomienda utilizar las siguientes funciones de utilidad:

```
def write(sock, text):
    print(f"write ({text})")
    data = text.encode()
    sock.sendall(len(data).to_bytes(2, 'big'))
    sock.sendall(data)

def read(sock):
    print("read()")
    bytes_to_read = int.from_bytes(sock.recv(2), 'big')
    chunks = []
    while bytes_to_read > 0:
        data = sock.recv(bytes_to_read)
        bytes_to_read -= len(data)
        chunks.append(data)
    return b"".join(chunks).decode()
```

La función *write(sock, text)* escribe en un socket el texto especificado. Para ello, primero envía el tamaño del mensaje, y después envía el mensaje. La función *read(sock)* efectúa la operación inversa, lee el texto escrito previamente. Para ello, primero lee el tamaño del mensaje y después lee el mensaje.

El esqueleto de la operación *serve(sock)* podría ser así (se muestra una posible implementación de la operación update como ejemplo):

```
def serve(self, sock):
  # se obtiene la petición en JSON y se descodifica
  req = json.loads(read(sock))
  try:
    if req[0] == "create":
      # servir la operación create
    elif req[0] == "describe":
      # servir la operación describe
    elif req[0] == "destroy":
      # servir la operación destroy
    elif req[0] == "insert":
      # servir la operación insert
    elif req[0] == "search":
      # servir la operación search
    elif req[0] == "update":
      # servir la operación update
    elif req[0] == "delete":
      # servir la operación delete
    else:
      # error
  except Exception as e:
    resp = ["ko", str(e)]
    write(sock, json.dumps(resp))
  sock.close()
```

Para arrancar el servidor sirviendo la base de datos “db” en el puerto 8000 de la máquina local se propone utilizar el siguiente comando:

```
> python3 server.py db localhost:8000
```

3. El cliente

Se implementará un cliente en el módulo *cli-remote.py*. A continuación se muestra un ejemplo de uso de dicha herramienta CLI:

```
> python3 cli-remote.py help
Usage: python3 cli-remote.py <host:port>
Available commands within the prompt
  help: show this message

  addcol <name> {key=val,}+: add a new collection with the specified schema
  rmcol <name>: remove de specified collection
  desc: describe the store

  add <col> {key=val,}+: add a new document with the specified attributes
  search <col> <query>: obtain all the documents which meet the specified query
  update <col> <query> {key=val,}+: update the documents meeting the query
  remove <col> <query>: remove the documents which meet the specified query
  exit: close the current connection

> python3 cli-remote.py localhost:8000
db > addcol users *email=str,name=str,*age=int
Col added.
db > add users email=a name=a age=18
Doc added with _id 2dc694b9-ead6-4805-9cfb-14fb00c47f1a.
db > add users email=b name=b age=18
Doc added with _id a2d3ae6c-44af-4fb5-8197-3a3f590ed8b1.
db > search users _id=2dc694b9-ead6-4805-9cfb-14fb00c47f1a
Results:
- id: 2dc694b9-ead6-4805-9cfb-14fb00c47f1a
  email: a
  name: a
  age: 18
db > search users age>10 age<20
Results:
- id: 2dc694b9-ead6-4805-9cfb-14fb00c47f1a
  email: a
  name: a
  age: 18
- id: a2d3ae6c-44af-4fb5-8197-3a3f590ed8b1
  email: b
  name: b
  age: 18
db > update users _id=2dc694b9-ead6-4805-9cfb-14fb00c47f1a email=c,age=19
Doc updated.
db > exit
Bye
```

Como se puede observar, funciona de igual manera que el módulo *cli.py* presentado en sesiones anteriores. Sin embargo, en este caso los comandos deben transmitirse al almacén de documentos a través de la red.

Por ello, en cada operación, se efectuará una conexión contra el servidor remoto, se enviará una petición y se esperará la respuesta. Para ello se pueden utilizar las operaciones *read()/write()* presentadas en el apartado anterior. A modo de ejemplo, se presenta una posible codificación de la operación *search:*

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
req = ["search", "users", {}]
write(sock, json.dumps(req))
status, resp = json.loads(read(sock))
sock.close()
if status != "ok": raise Exception(resp)
else: print(resp)
```

4. Replicación

El último paso consiste en extender el almacén de documentos para que sea replicado. Esto implica resolver fundamentalmente dos tareas:

1.- Protocolo de membresía: es necesario diseñar un protocolo que permita configurar los miembros de un grupo de nodos (las altas y las bajas). Esto servirá para conocer la lista de seguidores que el nodo líder posee.

2.- Protocolo de replicación: una vez se dispone de un grupo de nodos, es necesario diseñar un protocolo que permita atender las peticiones de lectura y escritura sobre el almacén.

4.1 Protocolo de membresía

La idea es diseñar un protocolo dinámico, donde los nodos se añaden y eliminan dinámicamente. Se sugiere implementar el siguiente mecanismo:

1. Primero arrancamos el almacén líder. Para ello, se podría utilizar el siguiente comando, donde se crea un servidor líder escuchando por el puerto 8000 y sirviendo el contenido del almacén en el path “db”.

```
> python3 server.py db localhost:8000
```

2. Arrancamos uno o varios seguidores. Para arrancar un seguidor, podríamos utilizar el siguiente comando. En este comando, se sirve el almacén en el path “db”, por el puerto 8001. Además, se especifica dónde está escuchando el líder (localhost:8000)

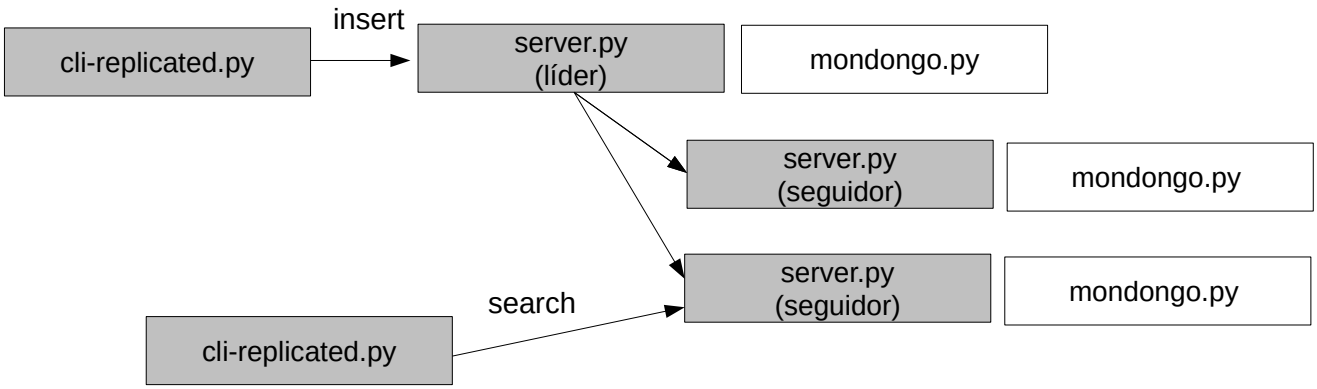
```
> python3 server.py db localhost:8001 localhost:8000
```

3. Entonces, el seguidor contactará con el líder a través de la red, indicándole que es un nuevo seguidor. El líder recibirá el mensaje y registrará al nuevo seguidor en su lista de seguidores activos.

4. La eliminación de miembros se producirá automáticamente. Cuando el líder intente conectar con un seguidor y reciba un error, lo eliminará automáticamente de su lista de seguidores activos.

4.2 Protocolo de replicación

Se propone utilizar la estrategia de replicación pasiva. En esta estrategia una réplica del almacén adopta el rol de líder. El resto de réplicas serán seguidoras. Las operaciones de escritura sólo pueden enviarse al líder. Las operaciones de lectura pueden enviarse tanto al líder como a los seguidores.



5. Fallos y recuperación

En un sistema replicado pueden suceder múltiples fallos. En esta sección se propone al alumno razonar sobre ello e implementar una estrategia que permita la detección de fallos y recuperación de nodos en caso de fallo.

A continuación se presentan distintos escenarios de fallos, así como sugerencias sobre las acciones a tomar para manejarlos.

Desde que se recibe una petición de escritura hasta que se escribe en todas las réplicas transcurre un tiempo significativo. Muchos fallos pueden suceder en este intervalo de tiempo: el líder puede fallar, el seguidor puede fallar.

Para resolver este problema, es necesario mantener un registro de todo lo que pasa y en qué punto falla la operación. Para ello es habitual utilizar un WAL (*write-ahead-log*) en cada nodo. De este modo, cuando el líder o el seguidor arranquen pueden recuperarse.

Si el líder falló, al arrancar debe revisar si dejó alguna operación de escritura a medias y retomarla donde la dejó.

Si falló el seguidor, entonces debe revisar la última operación que recibió y solicitar el resto de operaciones perdidas al líder.

Cuando un nuevo seguidor se une al sistema, es necesario transferirle las operaciones de actualización perdidas, o todas, si es la primera vez que se une al sistema.

Cuando un nodo líder falla, otro nodo debe tomar el liderazgo.