



Programación Distribuida y
Aplicada

Proyecto Final

Pichón Ramirez Jesús	201931546
Rivera Polvo Fernando	201942589
Vázquez Gabriel Luis Ángel	201957900
Vázquez Zamora Gabriel	201958021
Xicale Cabrera Irvyn	201963582



BENEMÉRITA
UNIVERSIDAD
AUTÓNOMA DE
PUEBLA



FACULTAD DE
CIENCIAS DE LA
COMPUTACIÓN

Contenido

Introducción.....	2
Diagramas generales de la implementación.....	3
Casos de Uso.....	4
Diagrama de Clases.....	5
Implementación en Python.....	7
Cliente.....	7
Servicio.....	8
Dockerizando.....	10
Dockerfile.....	11
Docker-compose.yml.....	12
requirements.txt.....	13
Conclusión.....	13

Introducción

El proyecto final se retoma desde el sistema de archivos que se implementó al principio del curso con funciones adicionales. Como primera función adicional está que el cliente sea dockerizado, lo que quiere decir que el programa pueda ejecutarse desde cualquier otro host. Además, las funciones que se hayan implementado se obtengan mediante llamadas RMI/RPC utilizando Pyro4.

El control de los archivos se mantiene desde MongoDB, donde se guardarán los datos de cada archivo, al mismo tiempo el servidor tendrá implementadas las operaciones que conforman CRUD donde cada cliente tendrá una copia en su base de datos local.

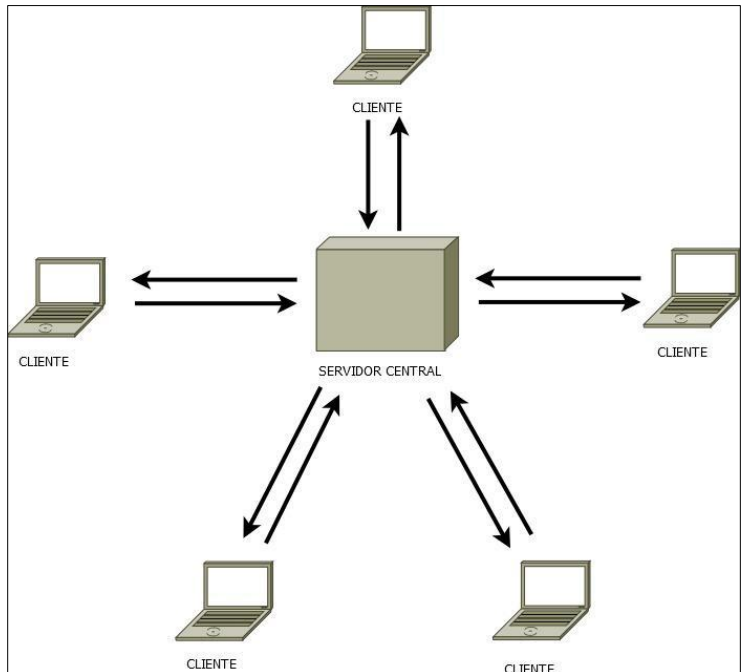
Los archivos se podrán transferir a través del servidor a los clientes conectados, cuando se edite un archivo, los cambios se harán también en el archivo original y mientras esté en uso no se podrá eliminar.

Una vez que se tienen los requerimientos adicionales para el proyecto, el siguiente paso es desarrollar diagramas que nos ayuden a visualizar lo que queremos implementar. Por lo cual, vamos a empezar a analizar los diagramas propuestos y revisaremos la implementación que se haya realizado.

Diagramas generales de la implementación

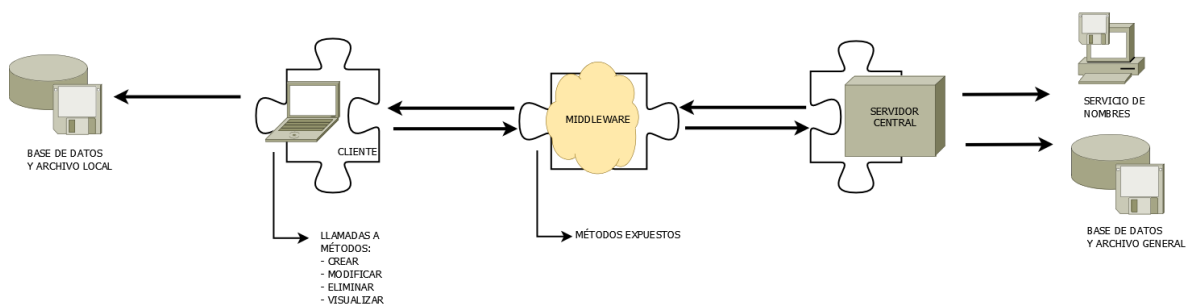
A partir de lo planteado, el primer punto a tener en cuenta es la topología que se tendrá al momento de integrar las máquinas que compartirán los archivos, estas actuarán como clientes y responderán las solicitudes que el servidor central reciba desde otros clientes.

La topología de estrella es el diseño de red por el que se optó trabajar, se ajusta a las necesidades que tenemos, además de que permite trabajar con un servidor central, que será el que tenga implementado el servicio de nombres y tenga su propia base de datos que guarde los archivos que se pidan a través de las peticiones de los clientes.



En el siguiente diagrama se analiza las funciones que tendrán implementadas los clientes y el servidor central, y como punto adicional se creará un middleware que contendrá los métodos que se vayan generando.

Desde el cliente se podrán generar las llamadas a los métodos que contiene CRUD (Create, Rename, Update, Delete), estas se envían a través del middleware hacia el servidor central, que tendrá el servicio de nombres junto a su base de datos que almacena los archivos que los clientes requieran. El cliente tendrá de igual forma una base de datos local.



Casos de Uso

El diagrama de casos de uso nos muestra los roles del sistema y cómo interactúan entre sí. En este caso solo se muestran al cliente y el servidor. Cabe aclarar que puede haber más de un cliente, teniendo en cuenta los recursos con los que el sistema distribuido cuente, habrá un máximo de estos mismos.

La interacción se basa prácticamente en el funcionamiento del servidor de nombres, compartiendo lista de archivos compartidos para que los clientes tengan acceso a todos los archivos en el sistema distribuido.

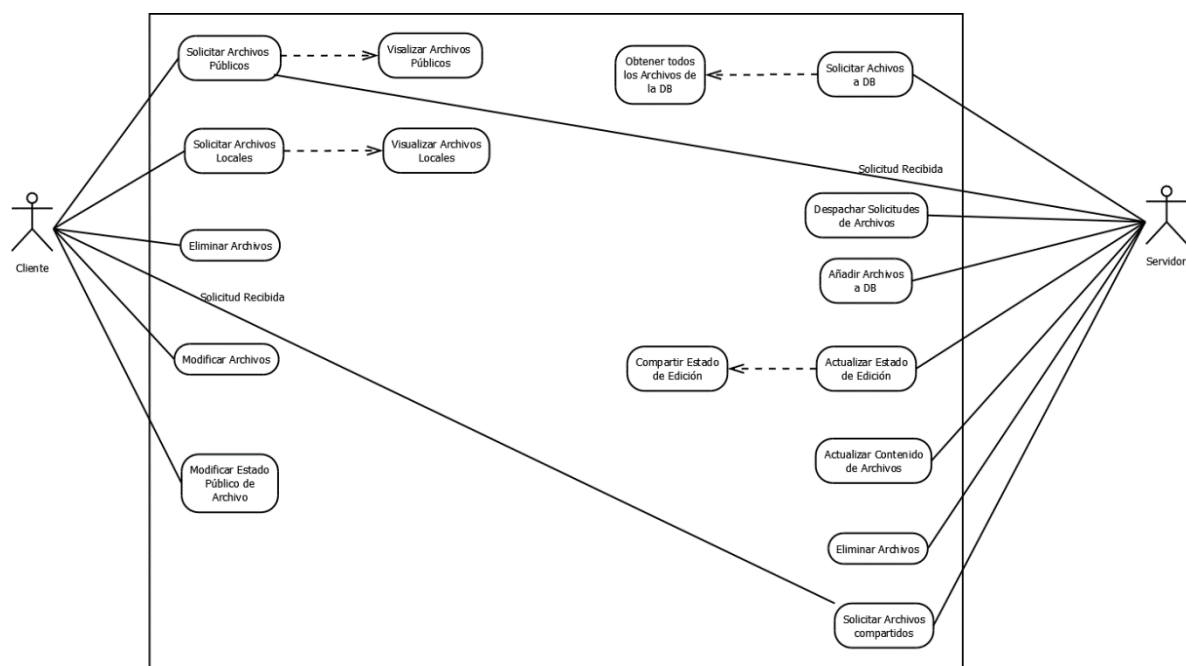
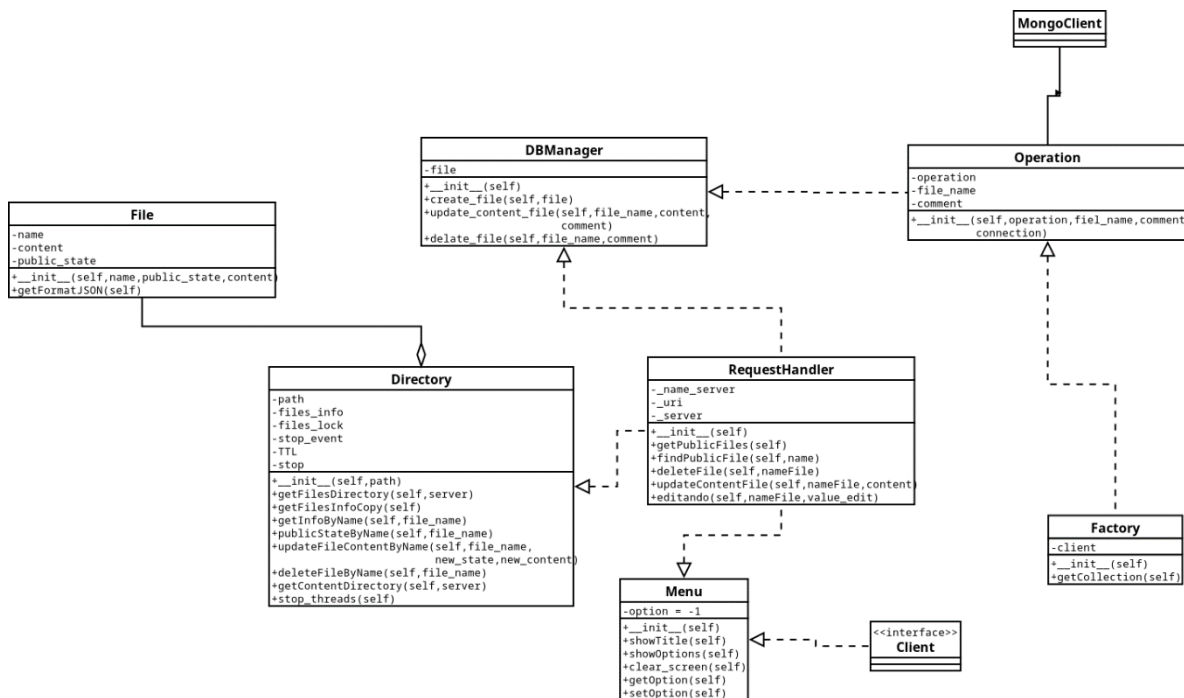


Diagrama de Clases

En este sistema realizamos implementaciones relacionadas con la gestión de archivos y operaciones en bases de datos MongoDB, organizadas a través de distintas clases y módulos para mejorar la modularidad y claridad del diseño. A continuación, se describen las características clave de cada diagrama.

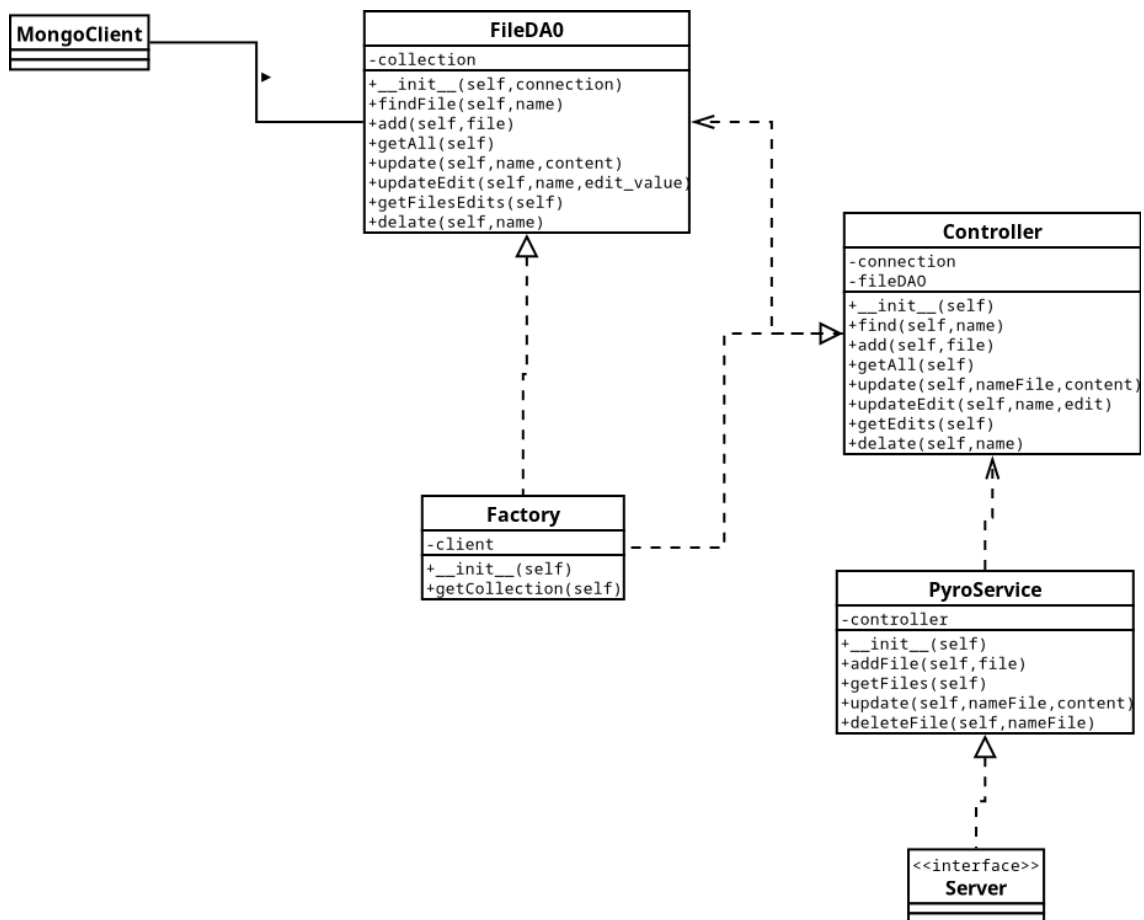
Cliente



Como se puede observar aquí, el componente principal de este sistema es el **requestHandler**, ya que es la clase que reúne e implementa las funcionalidades de las demás, o al menos de la mayoría. Podemos ver que hay una especial relación entre esta clase y la de **"directory"**, ya que a través de esta es como logramos obtener la información de los archivos publicados. Sin embargo, no podemos pasar por alto las siguientes tres clases **Factory**, **Operation**, y **DBManager**. Aquí, la relación entre **Factory** y **Operation** indica una dependencia directa, mientras que **DBManager** utiliza instancias de **Operation** para realizar operaciones de registro en la base de datos MongoDB. Esto también es importante debido a que sin estos módulos no podríamos tener una conexión a la base de datos y por ende las operaciones no podrían ser realizadas.

Servidor

Aquí, nuestra atención se centra en las clases “**FileDAO**” y “**Controller**”, ya que la relación entre estas es muy dependiente debido a que “**FileDAO**” le permite a “**Controller**” interactuar con la base de datos a modo de esta poder realizar las operaciones que la clase “**PyroService**” expone a los clientes, tales como, agregar, encontrar, actualizar y borrar. Además es importante observar que nuevamente tenemos una clase “**Factory**” que nos ayuda a establecer conexión con el cliente de MongoDB.



Implementación en Python

Dentro del proyecto general en Python, se implementó lo antes mencionado y se va a revisar el funcionamiento de cada parte.

Cliente

Hay 4 archivos que conforman el funcionamiento del cliente:

1. **cliente.py:**

Se visualiza el menú desde la terminal y se gestiona lo que se desee hacer hacia la base de datos.

Comenzamos con la importación de las clases connection, directory, menu y request_handler, el módulo threading que nos permitirá trabajar con hilos.

La configuración de objetos: creamos un controller para la base de datos, un menú y un directorio que será el que se va a compartir con los demás clientes.

Creación de hilo: Se crea un hilo que ejecuta el método '*get_files_directory*' cuya función es obtener los archivos del directorio que se comparte.

Interacción del usuario: Se muestran las opciones a elegir del menú y se guarda la opción elegida por el usuario.

2. **connection.py**

Se encarga de gestionar la conexión hacia MongoDB, es importante mencionar que aquí solo se guardan los registros de lo que se ha hecho en el directorio local de cada cliente, no de todos.

Clase 'Factory': Inicializa la instancia de la clase y crea un cliente de MongoDB. Con 'getCollection' se devuelven los archivos que la base de datos local tenga almacenados.

Clase 'Operation': Esta clase contiene la información sobre cada operación que se realice, contiene la hora, la operación, el nombre de archivo y un comentario adicional.

Clase 'DBManager': Contiene los métodos para realizar las operaciones dentro de la base de datos, según sea el caso, se crean, se actualizan y se borran archivos en la base de datos.

3. **directory.py**

Aquí se gestionan los archivos del directorio con la ayuda de hilos.

Clase 'File': Esta clase crea un archivo con atributos que después genera un formato de JSON. Los atributos son nombre, estado público y contenido.

Clase 'Directory': Gestiona el directorio con hilos para realizar operaciones que se requieran. Inicializa la clase con el *path* del directorio, una lista para almacenar información de los archivos (*files_info*), un semáforo para controlar el acceso a la lista de archivos (*files_info*), un evento de parada (*stop_event*) y un tiempo de vida que se configura en 10 segundos. Se define la función 'get_files' que devuelve los archivos de *files_info* la función 'get_files_directory_thread' que crea un hilo que actualiza los archivos en el directorio compartido cada 10 segundos, la función 'print_files_thread' que crea un hilo que imprime la información sobre los archivos cada 30 segundos y una función 'stop_threads' que establece un evento de parada para detener los hilos.

4. **menu.py**

Presenta las opciones del menú y gestiona la interacción del usuario con el sistema de archivos.

Clase 'Menú': Genera el menú con las opciones que puede realizar y se devuelve la opción elegida.

5. **request_handler.py**

Se encarga de actuar como cliente para interactuar con el servicio remoto de gestión de archivos.

En el constructor, la clase inicializa el cliente pyro. Localiza el servidor de nombres, obtiene la URL del servicio de archivos y crea un proxy para el servicio.

Se llama al método 'getPublicFiles' del servicio de archivos para obtener la lista de archivos públicos. De igual forma se hace la llamada a los métodos delete y update del servicio para hacer lo que el usuario haya definido para los archivos.

Servicio

Contiene 2 archivos que se encargan de la gestión de archivos.

1. **connection.py**

Este archivo se encarga de la conexión hacia MongoDB, esta será la base de datos general que guardará los archivos públicos de todo el sistema de archivos, es decir, de los archivos que cada cliente tenga como públicos.

Clase 'Factory': Es la clase encargada de crear conexiones hacia la base de datos de MongoDB, la función 'getCollection' retorna los archivos que haya dentro de la base.

Clase 'FileDAO': Se encarga de definir el comportamiento de los métodos de la clase controller. Contiene métodos que trabajan con los archivos, 'add' inserta un archivo y devuelve un ID asociado, 'getAll' devuelve todos los archivos de la colección, 'update' actualiza el contenido de un archivo y 'delete' borra el archivo seleccionado.

Además, se crean métodos para editar el parámetro edit (cuando el archivo se está editando), que otorga un valor de verdadero si es que se está editando.

Clase 'Controller': Define las funciones para la interacción de la base de datos. Aquí se hacen los llamados a los métodos de la clase anterior para realizar lo requerido en los archivos en la base de datos.

2. service.py

Aquí se implementa un servidor de Pyro que proporciona un servicio remoto para la gestión de los archivos.

Clase 'PyroService': Esta clase define métodos expuestos mediante pyro4 para gestionar archivos de manera remota, los métodos expuestos se definen para hacer las operaciones que ya revisamos anteriormente llamando a los métodos originales del controlador.

Después, se implementa la clase principal que inicializa el servidor pyro, crea un objeto 'PyroService' y registra sus métodos expuestos.

'Pyro4.Daemon': Crea un demonio que escuchará las llamadas remotas en la dirección y puerto de cada cliente.

'Pyro4.LocateNS': Localiza el servidor de nombres en la dirección y puerto especificado.

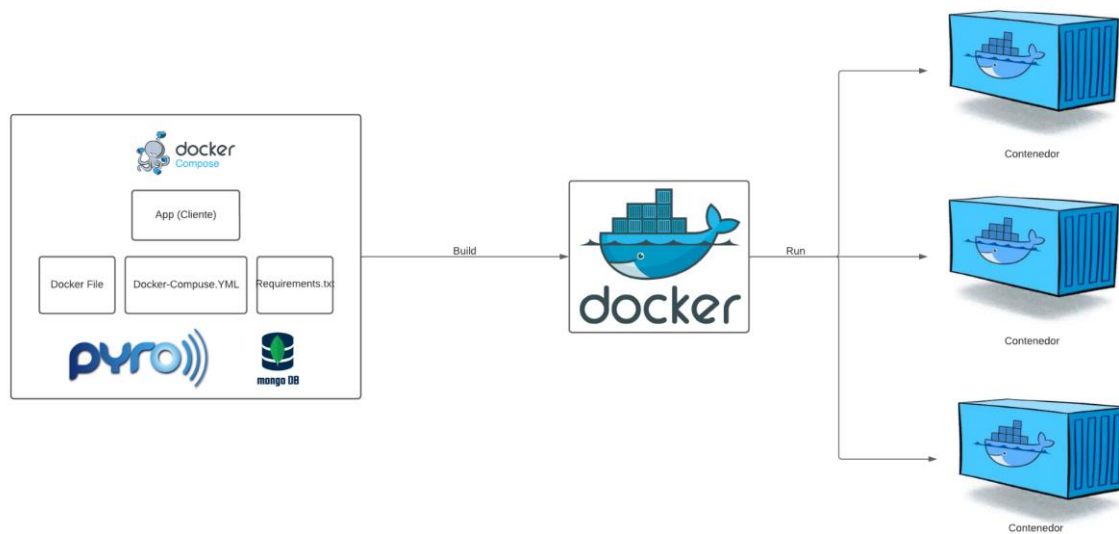
'daemon.requestLoop': Inicia el bucle del demonio para esperar llamadas remotas.

Dockerizando

Una vez teniendo nuestros archivos principales que componen nuestro proyecto, parecería que hemos terminado, sin embargo, todavía falta un paso importante para cumplir el objetivo de este proyecto y es que si se quiere tener realmente un sistema de archivos distribuido, es necesario que nuestra aplicación pueda ser ejecutada en cualquier computadora de nuestra red.

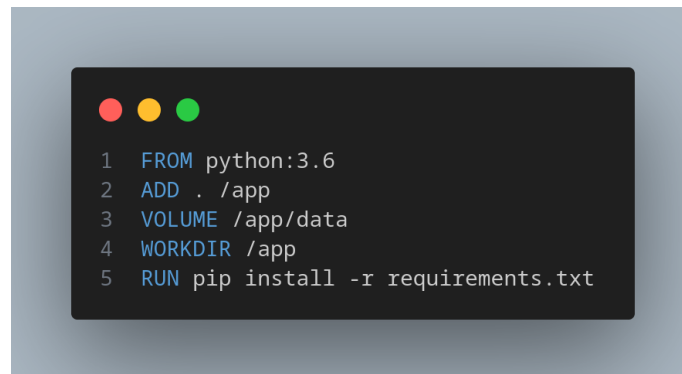
En esta parte nos apoyaremos de una herramienta muy poderosa como lo es docker, el cual nos ayudará a crear un contenedor para el proyecto donde podremos especificar los requerimientos necesarios para la ejecución de la aplicación y de esta forma podrá funcionar en cualquier entorno compatible con docker.

A continuación simplemente mostraremos la configuración que nosotros hicimos para la dockerización de nuestro proyecto.



Dockerfile

El archivo Dockerfile es utilizado en Docker para definir la configuración y los pasos necesarios para construir una imagen de contenedor. Cada línea en el Dockerfile representa una instrucción que Docker seguirá para construir la imagen.



```
1 FROM python:3.6
2 ADD . /app
3 VOLUME /app/data
4 WORKDIR /app
5 RUN pip install -r requirements.txt
```

Docker-compose.yml

El archivo docker-compose.yml es utilizado por Docker Compose para definir y gestionar aplicaciones Docker multi-contenedor. Este archivo permite describir todos los servicios, configuraciones y relaciones entre contenedores necesarios para ejecutar una aplicación.

```
1  version: '3'
2
3  services:
4    web:
5      stdin_open: true
6      build: .
7      command: ["python", "-u", "cliente.py"]
8      ports:
9        - "5000:5000"
10     volumes:
11       - C:\Users\lirvyn\OneDrive\Documents\programDistribuida:/app/data
12       - ../app
13     depends_on:
14       - db
15     networks:
16       - my_network
17
18   db:
19     image: mongo:4.4
20     hostname: test_mongodb
21     environment:
22       - MONGO_INITDB_DATABASE=db_files
23       - MONGO_INITDB_ROOT_USERNAME=root
24       - MONGO_INITDB_ROOT_PASSWORD=pass
25     ports:
26       - "27017:27017"
27     networks:
28       - my_network
29
30   networks:
31     my_network:
32       driver: bridge
33
```

requirements.txt

El archivo requirements.txt se utiliza para especificar las dependencias del proyecto. Cuando construyes una imagen de Docker que contiene una aplicación escrita en Python, docker instalará dentro del contenedor las dependencias con las que la app funciona correctamente.



Conclusión

El proyecto ha evolucionado para cumplir con requisitos adicionales, como la dockerización del cliente y el acceso remoto mediante llamadas RPC utilizando Pyro4. Se ha logrado una arquitectura robusta y escalable que permite a los clientes interactuar con archivos compartidos de manera distribuida. La integración con MongoDB asegura una gestión eficiente de los archivos y sus operaciones.

Además, la integración de Docker en el desarrollo fue crucial para la creación de entornos de ejecución coherentes y portables. El archivo docker-compose.yml nos ayudó a definir la configuración de dos servicios: el cliente y la base de datos (db). Esto facilita la replicación del entorno en diferentes máquinas y asegura la consistencia en la ejecución del sistema distribuido.

La aplicación de Docker Compose simplifica la orquestación de múltiples contenedores, estableciendo las dependencias y redes necesarias. La construcción y ejecución de contenedores se realiza de manera coordinada, lo que facilita el desarrollo, la prueba y la implementación en entornos distribuidos.

En resumen, el sistema de archivos distribuido implementado demuestra una solución completa y funcional para la gestión colaborativa de archivos en entornos distribuidos.