Actividad 7: Repaso concurrencia, paralelismo, y el uso de los módulos Python y Docker.

Para esta actividad en computación paralela y distribuida, nos vamos a enfocar en varios ejercicios prácticos y teóricos, enfocados en los temas de concurrencia, paralelismo, y el uso de los módulos Python y Docker.

Parte 1: Teoría

1. ¿Cuáles son las diferencias entre concurrencia y paralelismo?. ¿En qué escenario es preferible usar concurrencia sobre paralelismo y viceversa?
2. Presenta un repaso sobre los módulos **multiprocessing**, **asyncio**, y **concurrent.futures**.
3. Explica cómo Docker puede ser usado para simular entornos distribuidos y la importancia de la contenerización en proyectos paralelos y distribuidos.

Cada grupo escogerá uno de los siguientes temas y presentará una exposición junto con las preguntas anteriores:

Ejercicio 1: Modelos de concurrencia en sistemas distribuidos

Investiga y comparar diferentes modelos de concurrencia aplicados en sistemas distribuidos.

Escribir un ensayo donde comparen al menos tres modelos de concurrencia que se utilizan en sistemas distribuidos, como el modelo de actores, CSP (Communicating Sequential Processes) y la concurrencia basada en eventos. Deberán discutir las ventajas, desventajas, y casos de uso típicos para cada modelo, citando ejemplos reales o hipotéticos.

Preguntas guía:

- ¿Cómo maneja cada modelo los problemas de concurrencia como el deadlock y el livelock?
- ¿Qué tipo de aplicaciones se beneficiarían más de cada modelo?
- ¿Cómo influyen estos modelos en el rendimiento y escalabilidad de una aplicación?

Ejercicio 2: Investiga sobre Docker y Kubernetes en la Gestión de cargas de trabajo distribuidas

Explora cómo Docker y Kubernetes facilitan la gestión de aplicaciones en entornos de computación distribuida.

Deberás investigar y presentar un informe sobre cómo Docker y Kubernetes pueden ser utilizados para orquestar y manejar aplicaciones distribuidas. Deben enfocarse en características como la escalabilidad, balanceo de carga, y la recuperación de fallos.

Preguntas guía:

- ¿Cómo facilitan Docker y Kubernetes la implementación y gestión de microservicios?
- Describa un escenario donde el uso de Kubernetes proporciona una ventaja significativa sobre el uso de Docker solo.
- Explica el rol de los volúmenes persistentes y las redes en Kubernetes cuando se manejan servicios distribuidos.

Ejercicio 3: Análisis crítico de la arquitectura de microservicios

Analiza los desafíos y beneficios de implementar una arquitectura de microservicios en grandes empresas.

Redacta un análisis crítico sobre la implementación de microservicios en un contexto empresarial. Deberán considerar tanto los aspectos técnicos como los operativos, incluyendo la seguridad, la monitorización, y la integración continua.

Preguntas guía:

- ¿Cuáles son los principales desafíos técnicos y organizativos al adoptar microservicios en una gran empresa?
- ¿Cómo contribuyen los microservicios a la agilidad y escalabilidad de los sistemas empresariales?
- Evalúa el impacto de la adopción de microservicios en el mantenimiento y la gestión de la deuda técnica.

Ejercicio 4: Patrones de diseño en sistemas concurrentes y distribuidos

Investiga y discute varios patrones de diseño utilizados en la programación concurrente y distribuida.

Prepara una presentación o seminario sobre patrones de diseño específicos que se utilizan frecuentemente en sistemas concurrentes y distribuidos, como el patrón de singletons distribuidos, el patrón de saga, o el patrón de circuit breaker.

Preguntas guía:

- Describa cómo cada patrón aborda un problema específico en sistemas distribuidos.
- ¿Qué patrones serían más adecuados para manejar fallos en un sistema altamente disponible?
- Proporciona ejemplos de cómo estos patrones se han implementado en aplicaciones comerciales conocidas.

Ejercicio 5: Estudio de scaso sobre la tolerancia a fallos en sistemas distribuidos

Analiza estrategias de tolerancia a fallos en sistemas distribuidos utilizando casos de estudio reales.

Selecciona y estudia uno o más sistemas distribuidos reales (como bases de datos distribuidas, sistemas de archivos distribuidos, o sistemas de procesamiento de datos en tiempo real) y analiza cómo implementan la tolerancia a fallos. Deberán considerar aspectos como replicación de datos, particionamiento, y manejo de errores.

Preguntas guía:

- ¿Qué técnicas de tolerancia a fallos se emplean en el sistema elegido y por qué son efectivas?
- ¿Cómo se manejan los fallos de red o de hardware sin afectar la disponibilidad del servicio?
- Discute las compensaciones entre consistencia, disponibilidad y partición de tolerancia (CAP Theorem) que enfrenta el sistema estudiado.

Ejercicio 6: Docker y Mmcroservicios: desafíos de seguridad

Explora y discute los desafíos de seguridad específicos asociados con la implementación de microservicios utilizando Docker.

Investiga los desafíos de seguridad inherentes a la arquitectura de microservicios y cómo Docker puede tanto mitigar como exacerbar estos problemas. Deberán explorar temas como el aislamiento de contenedores, la gestión de secretos, y las políticas de red.

Preguntas guía:

- ¿Cuáles son los principales riesgos de seguridad al utilizar contenedores para microservicios y cómo se pueden mitigar?
- Explica la importancia de la gestión de secretos en un entorno de microservicios.
- ¿Cómo pueden las políticas de red y las herramientas de orquestación como Kubernetes ayudar a mejorar la seguridad de los microservicios?

Ejercicio 7: Performance y escalabilidad en Kubernetes

Investiga cómo Kubernetes maneja la performance y la escalabilidad en aplicaciones distribuidas de gran escala.

Deberás investigar y escribir un informe sobre los mecanismos que Kubernetes utiliza para mejorar la performance y permitir la escalabilidad de las aplicaciones que orquesta. Deberán considerar aspectos como la programación de pods, balanceo de carga, y autoescalado.

Preguntas guía:

- ¿Cómo afectan las decisiones de programación de pods a la performance general de una aplicación en Kubernetes?
- Describe el proceso de autoescalado en Kubernetes y cómo contribuye a la gestión eficiente de recursos.

- Analiza el impacto del balanceo de carga en la escalabilidad y la disponibilidad de los servicios.

Ejercicio 8: Principios de diseño de sistemas distribuidos

Examina y discute los principios fundamentales de diseño en la creación de sistemas distribuidos robustos y eficientes.

Debes escribir un ensayo donde discutan los principios de diseño esenciales para los sistemas distribuidos, como transparencia, escalabilidad, seguridad y manejo de fallos. Deberán proporcionar ejemplos de cómo estos principios se aplican en sistemas conocidos.

Preguntas guía:

- ¿Qué es la transparencia en un sistema distribuido y por qué es importante?
- ¿Cómo se puede lograr la escalabilidad en sistemas distribuidos sin comprometer la seguridad?
- Discute cómo el diseño de sistemas distribuidos puede influir en la facilidad de manejo de fallos.

Parte 2: Ejercicios prácticos

4. **Ejercicio con multiprocessing**: implementar un programa que use el módulo **multiprocessing** para calcular el factorial de números grandes de manera paralela.
5. **Ejercicio con asyncio**: crear un pequeño scraper de web que use **asyncio** para hacer múltiples solicitudes HTTP de manera concurrente.
6. **Ejercicio con concurrent.futures**: desarrolla un programa que use **ThreadPoolExecutor** para aplicar un filtro a varias imágenes simultáneamente.

Parte 3: Integración con Docker

7. **Contenerización de una aplicación**:
    - Creaa un Dockerfile para alguna de las aplicaciones desarrolladas en los ejercicios anteriores.
    - Explicación y ejecución de la aplicación dentro de un contenedor Docker.

Parte 4: Subida a github y revisión

8. **Git y GitHub**:
    - Cada grupo deberá subir su código a un repositorio de GitHub. Incluir un **README.md** que explique el propósito de cada script y cómo ejecutarlo.
9. **Revisión entre pares**:

- Cada grupo revisará el trabajo de otro grupo y proporcionará feedback a través de Issues en GitHub.

Ejemplos (replica esto)

Ejercicio con multiprocessing: Cálculo del factorial de números grandes

Utiliza el módulo **multiprocessing** de Python para calcular el factorial de varios números grandes en paralelo. Este ejercicio muestra cómo distribuir tareas intensivas entre diferentes procesos para acelerar la computación.

Pasos:

1 . Importa el módulo necesario:

```
import multiprocessing

import math
```

2 . Define la función que calcula el factorial:

```
def calcular_factorial(numero):

    resultado = math.factorial(numero)

    print(f"El factorial de {numero} es {resultado}")

    return resultado
```

3. Crea una lista de números grandes:

```
numeros = [100000 + x for x in range(5)]  # Lista de cinco números grandes
```

4. Distribuye la tarea entre múltiples procesos:

```
if __name__ == '__main__':

    with multiprocessing.Pool(processes=5) as pool:

        resultados = pool.map(calcular_factorial, numeros)

    print("Resultados:", resultados)
```

Explicación:

- Este código utiliza un **Pool** de procesos de **multiprocessing** para distribuir el cálculo del factorial entre varios procesos. Cada proceso ejecuta independientemente la función **calcular_factorial** y devuelve el resultado.
- Utilizando el método **map** del **Pool**, se distribuyen los números de la lista a diferentes procesos automáticamente.

Ejercicio con asyncio: Scraper web concurrente

Implementa un scraper web que utilice **asyncio** para realizar solicitudes HTTP concurrentemente a varias páginas web.

Pasos:

1 . Importa los módulos necesarios:

```
import asyncio

import aiohttp
```

2. Define la función asincrónica para hacer solicitudes HTTP:

```
async def fetch(url, session):
    async with session.get(url) as response:
        print(f"Status: {response.status}")
        data = await response.text()
        print(f"Data from {url} fetched")
        return data
```

3. Crea una tarea asincrónica principal que maneje múltiples solicitudes:

```
async def main(urls):
    async with aiohttp.ClientSession() as session:
        tasks = [fetch(url, session) for url in urls]
        await asyncio.gather(*tasks)
```

4. Ejecutar el programa asincrónico:

```
urls = ["http://example.com"] * 5  # Lista de URLs a consultar
```

```
asyncio.run(main(urls))
```

Explicación:

- Este código utiliza **aiohttp** para hacer solicitudes HTTP de manera asincrónica. La función **fetch** maneja la solicitud y procesamiento de la respuesta.
- **asyncio.gather** se utiliza para iniciar todas las tareas definidas en paralelo.

Ejercicio con concurrent.futures: Aplicar filtros a imágenes

Desarrolla un programa que use **ThreadPoolExecutor** de **concurrent.futures** para aplicar un filtro a varias imágenes simultáneamente.

Pasos:

1 . Importa los módulos necesarios:

```
from concurrent.futures import ThreadPoolExecutor

from PIL import Image, ImageFilter
```

2 . Define la función para aplicar el filtro:

```
def aplicar_filtro(imagen_path):
    img = Image.open(imagen_path)
    img = img.filter(ImageFilter.GaussianBlur(5))
    img.save(f"{imagen_path}_blurred.jpg")
    print(f"Filtro aplicado a {imagen_path}")
```

3 . Crea un pool de threads y distribuir las tareas:

```
imagenes = ["image1.jpg", "image2.jpg", "image3.jpg"]

with ThreadPoolExecutor(max_workers=3) as executor:
    executor.map(aplicar_filtro, imagenes)
```

Explicación:

- Este código usa **ThreadPoolExecutor** para crear un pool de hilos. Cada hilo toma una imagen de la lista y le aplica un filtro de desenfoque gaussiano.

- Utilizando **map**, las imágenes se procesan en paralelo, lo que puede ser significativamente más rápido que procesarlas de forma secuencial, especialmente para un conjunto grande de imágenes.

Contenerización de una aplicación

Crea un contenedor Docker para ejecutar una de las aplicaciones desarrolladas en los ejercicios anteriores, específicamente el scraper web concurrente realizado con **asyncio**.

Pasos para dockerizar la aplicación:

1 . Escribe el código de la aplicación:

- Asegúrate de tener un script Python (por ejemplo **scraper.py**) que implemente el scraper web concurrente.

2 . Crea un archivo Dockerfile:

- El **Dockerfile** contiene las instrucciones necesarias para construir la imagen Docker.

  # Utilizar una imagen base de Python oficial

  FROM python:3.9-slim

  # Establecer el directorio de trabajo en el contenedor

  WORKDIR /app

  # Copiar el script de Python en el contenedor

  COPY scraper.py .

  # Instalar aiohttp, necesario para el script

  RUN pip install aiohttp

  # Comando para ejecutar el script cuando el contenedor se inicie

  CMD ["python", "scraper.py"]

3 . Construye la imagen Docker:

- Abre una terminal y navega hasta el directorio donde se encuentra el **Dockerfile**.
- Ejecuta el siguiente comando para construir la imagen Docker:

docker build -t async-scraper .

4 . Ejecuta el contenedor:

- Una vez construida la imagen, ejecuta el contenedor utilizando el comando

  docker run async-scraper

Explicación de los componentes del Dockerfile:

- **FROM python:3.9-slim**: Esta línea especifica la imagen base. Aquí, se usa una versión "slim" de Python 3.9, que es ligera y suficiente para la mayoría de las aplicaciones Python.
- **WORKDIR /app**: Establece el directorio de trabajo dentro del contenedor. Todos los comandos siguientes se ejecutarán en este directorio.
- **COPY scraper.py .**: Copia el archivo **scraper.py** del directorio local al directorio de trabajo en el contenedor.
- **RUN pip install aiohttp**: Instala la biblioteca **aiohttp**, que es necesaria para el script de scraping.
- **CMD ["python", "scraper.py"]**: Define el comando predeterminado que se ejecutará cuando el contenedor se inicie, que en este caso es ejecutar el script **scraper.py**.

Beneficios de Dockerizar la aplicación:

- **Consistencia**: Docker asegura que la aplicación se ejecute en un entorno idéntico, independientemente del sistema operativo o de las configuraciones locales, lo que reduce los problemas de "funciona en mi máquina".
- **Seguridad**: Los contenedores proporcionan aislamiento de procesos, limitando la superficie de ataque en comparación con la ejecución de aplicaciones directamente en el host.
- **Portabilidad**: Una vez dockerizada, la aplicación puede ser desplegada fácilmente en cualquier sistema que soporte Docker, incluyendo plataformas en la nube, lo que facilita el escalado y la administración.

Subida a gitHub y revisión

Pasos para subir el código a un repositorio de GitHub:

1 . Crea un nuevo repositorio en GitHub:

- Cada grupo deberá acceder a GitHub y crear un nuevo repositorio haciendo clic en el botón "New" en la esquina superior derecha de la página de inicio.
- Nombrar el repositorio de manera descriptiva, por ejemplo, **async-scraper-project** o **parallel-image-processing**.

2 . Inicializa un repositorio local y conectarlo a GitHub:

- En la terminal o línea de comandos, navegar al directorio donde está el código del proyecto y ejecutar:

  git init

  git remote add origin URL_DEL_REPOSITORIO

  Reemplaza el **URL_DEL_REPOSITORIO** con la URL que GitHub proporciona después de crear el repositorio.

3 . Agrega los archivos al repositorio local y hacer el primer commit:

- Agrega todos los archivos relevantes al staging area con:

  git add .

- Realiza el primer commit

  git commit -m "Initial commit, adding project files"

Push al repositorio de GitHub:

- Sube el código al repositorio de GitHub

  git push -u origin main

  Si se solicita, ingresar las credenciales de GitHub.

Crea un archivo README.md:

- Es importante incluir un archivo **README.md** en la raíz del proyecto que explique el propósito de cada script y cómo ejecutarlo. Ejemplo de contenido para **README.md**:

  # Async Scraper Project

  Este proyecto contiene un scraper web implementado con asyncio y aiohttp en Python

  ## Cómo ejecutar

  Para ejecutar el scraper, asegúrese de tener Python y aiohttp instalados, y luego ejecute: python scraper.py

Más ejercicios

Escalado de aplicaciones con concurrent.futures

Utiliza **concurrent.futures** para mejorar el rendimiento de una tarea computacionalmente intensiva.

Descripción:

- Cada grupo desarrollará un script en Python que realice cálculos intensivos (por ejemplo, simulación de Monte Carlo, búsqueda de números primos grandes).
- El script debe usar el módulo **concurrent.futures** para paralelizar los cálculos y comparar el rendimiento con y sin paralelización.

Entregables:

- Código fuente del script.
- Un informe breve en **README.md** que compare el rendimiento y discuta las mejoras observadas.

Contenedorización de una Aplicación Python con Docker

Dockeriza una de las aplicaciones desarrolladas en los ejercicios anteriores.

Descripción:

- Elige una aplicación de los ejercicios anteriores (por ejemplo, el servidor web de **asyncio**).
- Crea un **Dockerfile** para contenerizar la aplicación.
- Documenta el proceso de construcción y ejecución del contenedor.

Entregables:

- **Dockerfile** y cualquier otro archivo de configuración necesario.
- **README.md** actualizado con instrucciones para construir y ejecutar el contenedor Docker.

Ejercicio: Desarrollo de un servidor web asíncrono con asyncio

Crea un servidor web asíncrono que sea capaz de manejar múltiples solicitudes de cliente simultáneas, servir archivos estáticos y realizar operaciones asíncronas en segundo plano.

Descripción

Implemeta un servidor web utilizando el módulo **asyncio** y **aiohttp** (una biblioteca que soporta cliente/servidor HTTP asíncrono) para manejar solicitudes HTTP de manera eficiente. El servidor deberá cumplir con los siguientes requisitos:

- **Servidor HTTP Asíncrono**: Capaz de recibir solicitudes HTTP y responder con contenido estático (como archivos HTML, CSS, imágenes).
- **Manejo de Rutas**: Implementar rutas específicas que ejecuten diferentes funciones, tales como servir una página de inicio, mostrar una lista de tareas en segundo plano y una página de estado del servidor.
- **Tareas Asíncronas en Segundo Plano**: Crear tareas que se ejecuten periódicamente en segundo plano, como simulaciones de tareas de mantenimiento o actualizaciones de estado.

Especificaciones técnicas

- **Estructura del servidor**: Utilizar **aiohttp** para crear el servidor y definir las rutas.
- **Rutas**:
  - **/**: Servir una página HTML simple que actúe como la página de inicio.
  - **/status**: Mostrar un JSON con el estado actual del servidor, incluyendo uptime y número de solicitudes manejadas.
  - **/background-tasks**: Mostrar una lista de tareas programadas y su estado actual.
- **Tareas de fondo**: Programar tareas como generación de logs simulados cada 10 segundos y simulación de operaciones de mantenimiento cada 5 minutos.

Tareas

1. **Configuración inicial**: Establecer el entorno de desarrollo, incluyendo la instalación de **aiohttp**.
2. **Implementación del servidor y rutas**: Usar **aiohttp.web** para configurar el servidor y definir las rutas necesarias.
3. **Manejo de solicitudes**: Implementar funciones asíncronas para manejar las solicitudes a las diferentes rutas.
4. **Servicio de archivos estáticos**: Configurar una ruta para servir archivos estáticos desde un directorio.
5. **Tareas asíncronas**: Usar **asyncio** para programar tareas que se ejecuten en intervalos regulares.
6. **Pruebas y validación**: Probar el servidor con múltiples clientes para asegurar que maneja las solicitudes de manera concurrente y correcta.

Entregables

- Código fuente del servidor en un repositorio de GitHub.
- Documentación en **README.md** que explique cómo ejecutar el servidor y una descripción de cada parte del código.