Universidad Técnica Estatal de Quevedo

Estudiante: Cedeño Orlando, Vilcacundo Jordy, Robalino Bryan y Orrala William

Curso: Ingeniería en Software 7mo “A”.

Tema: Arquitecturas Distribuidas.

Asignatura: Aplicaciones Distribuidas.

Docente: Guerrero Ulloa Gleiston Cicerón.

Índice

1. Introducción.....5
2. Arquitecturas Distribuidas 6
2.1 Definición y características.....6
2.2 Ventajas y desventajas. 7
2.2.1 Ventajas 7
2.2.2 Desventajas 7
3. Arquitecturas de Aplicaciones 7
4. Arquitecturas Web.....8
4.1.1 Cliente-servidor.....8
4.1.2 Arquitectura de tres capas. 9
4.1.3 Arquitectura RESTful y SOAP. 10
5. Desafíos de mantenimientos en sistemas distribuidos 12
6. Arquitecturas de blockchain y distribución de datos 13
7. Aplicaciones prácticas de arquitecturas distribuidas en empresas 15
7.1 Aplicación práctica de enfoques heurísticos para mejorar la arquitectura de red de una empresa de mensajería.....15
7.2 Una encuesta sobre la computación distribuida en flujo 15
7.3 Vientos oceánicos de alta resolución: infraestructura de nube híbrida para el procesamiento de imágenes satelitales.....16
8. Estrategias de Despliegue para sistemas Distribuidos 16
8.1 Recommending Deployment Strategies for Collaborative Tasks 16
8.2 Estrategia de Implementación de Contenedores en Redes de Borde 17

8.3 Investigación y Diseño de Estrategia Dinámica Arquitectura de Control Distribuido en Power Internet de las Cosas 17

8.4 Hacia una estrategia de movimiento óptimo distribuido para la recopilación de datos en redes de sensores inalámbricos 18

9. Consistencias y Coherencias en sistemas distribuidos 18

9.1 Consistencias en sistemas distribuidos 18

9.2 Coherencia en sistemas distribuidos 20

10. Desafíos de rendimiento en arquitecturas distribuidas 22

10.1 Latencia de red 22

10.2 Ancho de banda 23

10.3 Consistencia y coherencia de datos 23

10.4 Distribución de carga 23

10.5 Tolerancia a fallos 23

10.6 Escalabilidad 23

10.7 Coordinación y sincronización 24

10.8 Caché distribuido 24

11. Tecnologías emergentes en sistemas distribuidos 24

11.1 Computación en la nube sin servidor (Serverless) 24

11.2 Blockchain y tecnologías de registro distribuido (DLT) 25

11.3 Microservicios y contenedores 25

11.4 Orquestación de contenedores 25

11.5 Procesamiento de streaming y análisis en tiempo real 25

11.6 Inteligencia Artificial (IA) distribuida 26

11.7 Redes 5G 26

11.8 Seguridad distribuida 26

11.9 Arquitecturas de nube híbrida y multi-nube 26

12. Integración de contenedores en arquitecturas distribuidas 26
12.1 Empaquetado de aplicaciones 27
12.2 Portabilidad 28
12.3 Aislamiento 28
12.4 Escalabilidad 28
12.5 Facilita la gestión 28
12.6 Control de versiones y reproducibilidad 28
12.7 Entornos de desarrollo consistentes 29
12.8 Integración con herramientas de CI/CD 29
12.9 Facilita la adopción de microservicios 29
13. Arquitectura microservicios vs monolíticas 29
13.1 Microservicios 29
13.1.1 Características 30
13.2 Monolíticos 30
13.2.1 Características 30
13.3 Automatización en implementaciones distribuidas 31
13.4 Herramientas de implementación continua (CI/CD) 31
13.5 Infraestructura como código 31
14. Desarrollo de aplicaciones escalables en entornos distribuidos 31
14.1 Impacto en la nube de las arquitecturas distribuidas 32
15. Practica 33
15.1 Problema 33
15.2 Descripción 33
15.3 Código paso a paso 34
15.4 Interfaces 42
16. Conclusión 43

17. Referencias 44

1. Introducción

Las Arquitecturas Distribuidas han surgido como un paradigma revolucionario en el continuo y vertiginoso avance de la tecnología, desafiando las estructuras convencionales y redefiniendo la forma en que concebimos y construimos sistemas informáticos. Este informe analizará minuciosamente el mundo fascinante de las aplicaciones distribuidas, un entorno donde la descentralización, la escalabilidad dinámica y la resiliencia se combinan para impulsar el progreso de la computación.

Las aplicaciones distribuidas se componen de sistemas interconectados que ayudan an orquestar una red compleja y sinérgica. Las siguientes secciones analizarán los principios fundamentales que sustentan estas arquitecturas. La descentralización, que mejora la adaptabilidad, y la escalabilidad, que permite el crecimiento orgánico, son algunas de las características clave que dan forma a la ingeniería detrás de las aplicaciones distribuidas. Además, abordaremos los problemas inherentes, como la gestión de la consistencia en un entorno distribuido y cómo garantizar la seguridad y la integridad de los datos.

Desde las estructuras monolíticas hasta el surgimiento de arquitecturas basadas en microservicios, observamos cómo la evolución tecnológica ha guiado la transición de sistemas centralizados a una red interconectada de nodos distribuidos. Este informe busca arrojar luz sobre el tapiz de tecnologías y conceptos que componen las Aplicaciones Distribuidas, explorando su impacto, sus fundamentos y las implicaciones que tienen en el diseño y desarrollo de sistemas informáticos modernos.

2. Arquitecturas Distribuidas

2.1 Definición y características.

La arquitectura de los sistemas distribuidos se caracteriza por sus elementos individuales y las conexiones entre ellos. El objetivo fundamental de una arquitectura es asegurar que la estructura satisfaga tanto las necesidades actuales como las potenciales en el futuro. Los aspectos primordiales que se tienen en cuenta abarcan la confiabilidad, la fiabilidad de administración, la capacidad de adaptación y la eficiencia económica del sistema [1] .

La arquitectura de software se constituye a partir de la disposición de los componentes de un programa o sistema, sus conexiones entre sí, y los principios y normas que dictan su diseño y desarrollo a lo largo del tiempo [2].

Algunas características cruciales de los sistemas distribuidos incluyen:

- Ocultamiento de las diferencias en la comunicación entre las diversas computadoras y la organización interna del sistema, proporcionando al usuario una experiencia transparente [3].
- Los sistemas distribuidos avanzados pueden abarcar una variedad de nodos, desde grandes computadoras de alto rendimiento hasta pequeñas computadoras enchufables o incluso dispositivos más pequeños. Un principio fundamental es que estos nodos pueden operar de manera independiente. Sin embargo, es evidente que, si se ignoran entre sí, no tiene sentido incluirlos en el mismo sistema distribuido [2].
- Un sistema distribuido debería aparecer como un sistema único y coherente. En algunos casos, los investigadores incluso han llegado a afirmar la necesidad de una vista de sistema única, lo que significa que los usuarios finales ni siquiera deberían notar que están lidiando con el hecho de que los procesos, datos y control están dispersos en una red de computadoras [2].

2.2 Ventajas y desventajas.

2.2.1 Ventajas

Compartición de datos: Facilita el acceso de varios usuarios a una base de datos o archivo compartido.

• Compartición de dispositivos: Posibilita compartir recursos costosos, como plotters o impresoras láser, entre diferentes usuarios.

• Comunicación: Ofrece la opción de comunicación directa de usuario a usuario mediante herramientas como telnet o correo electrónico, entre otras.

• Confiabilidad: Permite distribuir la carga de trabajo entre distintas computadoras según sus funciones y capacidades, lo que proporciona mayor flexibilidad y confiabilidad al sistema [4].

2.2.2 Desventajas

• Software: Una gran parte del software diseñado para sistemas distribuidos aún está en desarrollo.

• Redes: Los problemas de transmisión en las redes de comunicación siguen siendo comunes, especialmente en la transferencia de grandes volúmenes de datos, como en el caso de archivos multimedia.

• Seguridad: Mejoras en los esquemas de protección son necesarias para garantizar un acceso más seguro a información confidencial o secreta.

• Tolerancia a fallas: Las fallas operativas y de componentes todavía ocurren con frecuencia, y es necesario abordar este aspecto para mejorar la estabilidad del sistema [1].

3. Arquitecturas de Aplicaciones

- Monolíticas vs. microservicios.
- Componentes y módulos.

4. Arquitecturas Web

4.1.1 Cliente-servidor.

El término "concepto cliente-servidor" se refiere a los principios arquitectónicos aplicados a diversos sistemas informáticos de laboratorio. En este concepto existen una o más máquinas centrales que actúan como servidores y varias estaciones de trabajo llamadas clientes, una ventaja es que permite una asignación eficiente de recursos.

El cliente puede utilizar los servicios proporcionados por el servidor. El servidor informático del laboratorio almacena sus datos en una base de datos a la que pueden acceder diferentes clientes en función de su autorización de acceso y proporciona programas, memoria, potencia informática y servicios de comunicación, la computadora cliente también es un sistema funcional sin servidor. su función es proporcionar a los usuarios una interfaz de usuario.

Un ejemplo es sobre la arquitectura de juegos cliente/servidor se refiere a una arquitectura distribuida típica para el soporte de juegos en red. En esta arquitectura, un solo nodo desempeña el papel del servidor, es decir, mantiene el estado del juego y se comunica con todos los demás nodos (los clientes). El servidor notifica los movimientos del juego generados por los jugadores y calcula las actualizaciones del estado del juego [5].

En este enfoque sobre la invocación asíncrona y su impacto en el rendimiento del sistema cliente-servidor. Los resultados muestran que los servidores con arquitecturas asíncronas basadas en eventos pueden experimentar un menor rendimiento debido a cambios de contexto intermedios y ciertas condiciones de carga y red.

Se propone una solución híbrida que utiliza múltiples arquitecturas asíncronas para adaptarse a los cambios de carga y red, logrando una mayor eficiencia frente a otro tipo de soluciones, servidores,

mejorando entre un 19% y un 90%, dependiendo de la carga de trabajo y estado de la red.

4.1.2Arquitectura de tres capas.

En la arquitectura en tres capas por ejemplo se destaca la importancia en relación con la reforma del sistema de salud. La arquitectura del sistema de seguro médico ha evolucionado desde un modelo cliente/servidor (C/S) a una arquitectura de tres niveles para satisfacer las necesidades de escalabilidad y mantenibilidad.

Se separa lógicamente las funciones del usuario, la lógica empresarial y el acceso a la información, y es un enfoque más avanzado y estructurado, se presenta requisitos estándar para los sistemas de gestión de seguros de salud y aboga por el uso de un modelo de tres partes para estadísticas sistemáticas [6].

Un ejemplo sobre la arquitectura en tres capas, se destaca la importancia de la personalización a gran escala en la construcción industrial para aumentar la eficiencia sin sacrificar la flexibilidad del diseño, Ofrece un configurador como plataforma, pero destaca la falta de integración con la cadena de suministro. Un ejemplo al aplicar es el marco conceptual de fabricación de "kits de piezas" utilizando modelos digitales predesarrollados.

El configurador fue prototipado utilizando una arquitectura de tres niveles, incorporando cada nivel de detalle del conjunto de piezas, se implementó en todas las etapas de la construcción, desde la planificación del sitio hasta la generación de modelos 3D, y sus beneficios han quedado demostrados a través del exitoso configurador de edificios modulares [7].

Este ejemplo es sobre implementar la gestión de relaciones con los clientes (CRM) en el sector financiero para mantener una ventaja competitiva a través de la integración y el intercambio de información. Se propone una arquitectura de tres capas de un sistema de

asesoramiento sobre gestión de activos que utiliza un enfoque de múltiples agentes.

Cada capa realiza una función específica, la primera capa está diseñada para satisfacer a diferentes clientes y brindar servicios personalizados, la segunda capa analiza información financiera y proporciona información, y la tercera capa organiza datos para análisis de segundo nivel [8].

4.1.3 Arquitectura RESTful y SOAP.

Las APIs REST, basadas en la estructura RESTful, simplifican la interacción entre clientes y servicios web mediante peticiones HTTP. Los usuarios envían solicitudes para acceder o modificar recursos manejados por el servicio, dirigidas a puntos de API identificados por rutas y métodos HTTP, como post, get, put y delete. Una vez procesada la solicitud, el servicio responde con un código de estado HTTP que refleja el resultado, ya sea éxito o error [9].

Un ejemplo es sobre un estudio que detalla la evaluación de 500 sitios web líderes según Alexa, que alegan ofrecer APIs de servicios web REST. Se destaca la prevalencia de soporte para JSON y la dependencia de documentación generada por software. A pesar de esta tendencia, se observa una diversidad significativa en la adherencia a las mejores prácticas, con solo el 0,8 por ciento cumpliendo estrictamente todos los principios REST [10].

Rest resalta su énfasis en interacciones sin estado y representación de recursos. La investigación, realizada a través de la implementación de REST y SOAP APIs mediante JAX-RS y JAX-WS, respectivamente, proporciona un análisis comparativo de características de la Interfaz de Programación de Aplicaciones (API), incluyendo tiempos de respuesta, uso de memoria y velocidad de ejecución [11].

La importancia central de REST como el protocolo estándar para servicios a través de APIs RESTful, subrayando desafíos en la prueba exhaustiva debido a la limitada disponibilidad de detalles internos. Se

advierte sobre riesgos de cambios imprevistos en las APIs que podrían resultar en fallos críticos con impactos financieros y de confianza. A pesar de avances en investigación sobre pruebas unitarias automáticas, se subraya la carencia de una visión global de las metodologías actuales [12].

En el contexto de Tapis, una plataforma que adopta la arquitectura de microservicios se centra en construir resiliencia y escalabilidad. Los servicios están diseñados para tener un acoplamiento flexible, facilitando el mantenimiento y permitiendo programaciones de desarrollo independientes. La interacción de usuarios y aplicaciones con Tapis se logra mediante solicitudes HTTP autenticadas a sus puntos finales públicos [13].

Un servicio SOAP es fundamental para que los servicios web a través de una aplicación puedan hacer uso de la información que nos brinda. La implementación de SOAP se ha realizado en varios lenguajes de programación, y su uso combinado de XML y HTTP proporciona un protocolo estándar para la comunicación cliente-servidor en Internet [14].

Este protocolo permite que los mensajes SOAP pasen por intermediarios en el camino hacia la computadora que gestiona el recurso, y los servicios de middleware pueden utilizar estos intermediarios para realizar procesamiento adicional [14].

La estructura incluye encabezados para datos de infraestructura y cuerpos para datos de aplicación. La gestión de mensajes se realiza a través de un "nodo SOAP" facilitado por plataformas de middleware, con reglas cruciales establecidas por el protocolo SOAP para el procesamiento adecuado, abordando aspectos como la interpretación de encabezados y la gestión de fallos. SOAP proporciona una base estandarizada para una comunicación eficiente entre sistemas distribuidos [15].

Los resultados de estudios muestran que estas técnicas tienen un impacto significativo en la mejora del rendimiento de SOAP. Lo más destacado es el uso eficiente de la funcionalidad SOAP, proporcionando sólo la funcionalidad necesaria y una serialización diferencial suficiente de mensajes XML. En conjunto, estas mejoras pueden mejorar la funcionalidad y usabilidad de SOAP en varios entornos, abordando problemas de rendimiento y mejorando su usabilidad en aplicaciones comerciales y científicas [16].

5. Desafíos de mantenimientos en sistemas distribuidos

Existe un aumento en la demanda de sistemas distribuidos complejos debido a la globalización. La importancia de abordar los problemas específicos de los sistemas distribuidos para minimizar fallas de rendimiento y mejorar la Calidad de Servicio (QoS).

El estudio, que revisa 12 temas, resalta la relevancia actual de aspectos como seguridad y privacidad, calidad de servicio, gestión de recursos y problemas de sincronización. Los desafíos en sistemas distribuidos evolucionan en cada etapa de desarrollo, como el diseño y el mantenimiento, y se adaptan a nuevos avances tecnológicos. Se gestionarán eficazmente los problemas en sistemas distribuidos durante el desarrollo y mantenimiento [17].

Se destaca la inevitabilidad de los cambios en los sistemas de software después de su implementación. se reconoce la aparición de nuevos requisitos y la necesidad de modificaciones para corregir errores o mejorar el rendimiento. Se usa el sistema CMMS para verificar el cumplimiento normativo de mantenimiento, pero se señalan desafíos al cubrir solicitudes de cambios. Además, se propone una forma y herramienta para rastrear y reportar solicitudes de mantenimiento, abordando eficazmente los desafíos en los sistemas CMMS y respaldando los requisitos de cambios de software [16].

Este enfoque aborda la necesidad de un mecanismo de mantenimiento de consistencia en aplicaciones Peer-to-Peer (P2P) debido a las frecuentes actualizaciones de datos. Se propone un esquema escalable y eficiente para sistemas P2P heterogéneos que aborda problemas encontrados en enfoques centralizados y descentralizados anteriores.

El esquema considera la heterogeneidad de los sistemas y organiza nodos de réplica en una estructura jerárquica consciente de la localidad. La construcción dinámica de un árbol de propagación de mensajes de actualización sobre la capa superior para optimizar la propagación de contenidos actualizados. Los resultados de análisis teóricos y simulaciones muestran que el enfoque propuesto reduce el costo en un rango significativo, entre el 25% y el 67%, en comparación con diseños anteriores [18].

Este ejemplo propone un ecosistema completo de IoT Big Data en una arquitectura de tres capas (edge, cloud y aplicación) para abordar la complejidad del proceso, desde la recopilación de datos hasta la visualización. La distribución eficaz de tareas entre las capas de nube y edge, junto con la introducción de un autoencoder asistido por edge computing, busca resolver desafíos de confiabilidad y escalabilidad.

La guía de implementación API y el estudio de caso real respaldan la propuesta, demostrando el rendimiento mejorado de sistemas de IoT basados en edge computing y aprendizaje profundo. Se centra en implementar eficazmente el mantenimiento predictivo en entornos IIoT mediante tecnologías como IoT, Big Data, inteligencia artificial y edge computing [7].

6. Arquitecturas de blockchain y distribución de datos

La naturaleza disruptiva de la tecnología blockchain, diseñada para implementar sistemas distribuidos y descentralizados de manera segura. Enfatiza la capacidad de compartir, almacenar y verificar datos transaccionales sin depender de una autoridad central de autenticación

o verificación. Aunque los sistemas basados en blockchain ofrecen varios componentes y variantes arquitectónicas para construir software seguro.

Existe una taxonomía de decisiones de diseño de arquitectura comunes en sistemas basados en blockchain. Cada una de estas decisiones se relaciona con posibles ataques de seguridad y las amenazas que representan. Se utiliza un enfoque de mapeo basado en las categorías de tácticas de ataque de MITRE y el modelado de amenazas STRIDE de Microsoft para clasificar sistemáticamente amenazas y sus ataques asociados [7].

Este ejemplo es sobre adopción generalizada de blockchain en varios sectores y enfatiza sus mecanismos descentralizados, persistencia y auditabilidad. Si bien los documentos de encuestas han explorado las tecnologías de blockchain desde diferentes ángulos, como las monedas digitales, los algoritmos de consenso y los contratos inteligentes, ha habido una falta de enfoque en los sistemas de gestión de datos de blockchain.

Se realizaron una encuesta exhaustiva que clasifica los mecanismos de administración de datos en tres capas: arquitectura de blockchain, estructura de datos de blockchain y motor de almacenamiento de blockchain. Cada capa se describe, cubriendo cómo se registran las transacciones, la estructura interna de los bloques y la forma de almacenamiento de datos [7].

Una arquitectura novedosa que combina características de blockchain, fog computing y cloud computing para gestionar datos de IoT. Destaca el uso de blockchain para establecer una red peer-to-peer distribuida donde participantes no confiables pueden interactuar sin intermediarios de confianza. La evaluación se centra en cómo este mecanismo aborda los desafíos de IoT en términos de accesibilidad a múltiples dispositivos.

La arquitectura propuesta incluye una capa de edge computing en presencia de blockchain. El uso de fog computing permite el análisis local de datos sensibles en lugar de enviarlos a la nube, y los nodos edge pueden monitorear y controlar dispositivos IoT. Se destaca que esta gestión puede optimizarse mediante la integración de Software Defined Network (SDN) y Network Functions Virtualization (NFV) para una administración eficiente de recursos [18].

7. Aplicaciones prácticas de arquitecturas distribuidas en empresas

7.1 Aplicación práctica de enfoques heurísticos para mejorar la arquitectura de red de una empresa de mensajería.

Se presenta dos estrategias para mejorar el rendimiento de la arquitectura de red de una empresa de mensajería. Se basan en un algoritmo heurístico para determinar el número, la ubicación y las propiedades de las estaciones de servicio de la red, y en un método recursivo vertical para implementar instalaciones de manera jerárquica en la arquitectura de red.

La implementación de estas estrategias permite mejorar el rendimiento en función de la ubicación, distribución y características específicas de cada instalación. Estas estrategias pueden ser una solución efectiva para optimizar el rendimiento de la arquitectura de red de una empresa de mensajería [19].

7.2 Una encuesta sobre la computación distribuida en flujo

Este tema aborda la creciente importancia del procesamiento de datos en tiempo real (streaming data) en el contexto del explosivo crecimiento de datos. Se exploran diferentes arquitecturas distribuidas, como Storm, Spark y Samza, aplicadas con éxito en la industria para procesar datos en tiempo real. Cada una de estas arquitecturas

presenta diseños únicos y prácticos adaptados a requisitos específicos en diversos entornos de cómputo distribuido. Aunque se reconocen problemas en ciertos aspectos, se destaca el éxito y el potencial de desarrollo futuro de estas arquitecturas [20].

7.3 Vientos oceánicos de alta resolución: infraestructura de nube híbrida para el procesamiento de imágenes satelitales

El ejemplo es sobre una aplicación práctica de tecnologías de computación en la nube para abordar desafíos en observación terrestre y meteorología. La arquitectura de la nube se vuelve esencial con servicios de pago por uso. Se resalta la relación entre la complejidad del procesamiento de datos satelitales y soluciones de la computación en la nube, enfocándose en acceso centralizado a datos, almacenamiento escalable, cómputo distribuido y optimización de hardware [21].

8. Estrategias de Despliegue para sistemas Distribuidos

8.1 Recommending Deployment Strategies for Collaborative Tasks

Este trabajo explora Estrategias de Despliegue para Sistemas Distribuidos, centrándose en ayudar a los solicitantes a implementar tareas colaborativas en la participación ciudadana.

Se estudia las estrategias de despliegue alineadas con parámetros deseados como calidad, latencia y restricciones de costos, implica decisiones en tres dimensiones: Estructura (solicitud secuencial o simultánea de la fuerza laboral), Organización (colaborativa o independiente) y Estilo (depender únicamente de la multitud o combinarlo con algoritmos de máquinas).

La solución propuesta, StratRec, es una capa intermedia impulsada por la optimización que proporciona recomendaciones considerando la disponibilidad de los trabajadores. Experimentos extensos en Amazon Mechanical Turk y experimentos sintéticos validan los aspectos cualitativos y de escalabilidad de StratRec [22].

8.2 Estrategia de Implementación de Contenedores en Redes de Borde

Este ejemplo destaca la importancia de la computación en el borde (edge computing) para aplicaciones sensibles a la latencia y señala las limitaciones de las herramientas comunes de orquestación de contenedores en términos de considerar la localidad durante el despliegue.

Se aborda esta limitación mediante la recopilación de datos en tiempo real sobre latencia y consumo de recursos, con el objetivo de optimizar las ubicaciones de despliegue según la demanda. La evaluación realizada en 16 regiones de AWS demuestra el éxito de la estrategia al reducir significativamente la latencia promedio en comparación con los algoritmos de despliegue estándar, mostrando un potencial impacto positivo en la eficiencia de las aplicaciones y servicios en el borde de la red [23].

8.3 Investigación y Diseño de Estrategia Dinámica Arquitectura de Control Distribuido en Power Internet de las Cosas

La información destaca el crecimiento rápido de la Internet de las Cosas (IoT) y la prevalencia de dispositivos terminales en sistemas de IoT de potencia. Se propone una arquitectura de control distribuido para abordar los requisitos de seguridad, permitiendo la construcción dinámica de estrategias y flexibilidad en la implementación. Ante los desafíos de la heterogeneidad de los terminales, la estrategia se divide en capas: estrategia, transporte y física.

Este enfoque establece un ciclo cerrado que ajusta dinámicamente dispositivos y datos, asegurando la seguridad en la producción de energía y permitiendo la gestión y control unificados de terminales heterogéneos a gran escala [24].

8.4 Hacia una estrategia de movimiento óptimo distribuido para la recopilación de datos en redes de sensores inalámbricos

El último ejemplo propone una estrategia de movimiento distribuido para colectores móviles en redes de sensores inalámbricos, considerando agentes físicos o paquetes de consulta. Al formular el problema como paseos aleatorios en un gráfico de nodos, aborda limitaciones de espacio de búfer y tasas de llegada de datos heterogéneas.

Identifica una estrategia óptima para minimizar la pérdida de datos en todos los nodos mediante enfoques markovianos. Los resultados demuestran que esta estrategia supera significativamente a un paseo aleatorio estándar, generando ahorros de costos del 70% en la implementación de múltiples colectores sin comprometer la tasa de pérdida de datos objetivo [25].

9. Consistencias y Coherencias en sistemas distribuidos

9.1 Consistencias en sistemas distribuidos

La consistencia en sistemas distribuidos se refiere a la propiedad de que todos los nodos en el sistema ven los mismos datos en el mismo orden. En otras palabras, la consistencia garantiza que las operaciones de lectura y escritura en los datos distribuidos se comporten de manera predecible y coherente para todos los nodos del sistema [15].

Según Ciciani B, et al [16], la consistencia es un concepto importante en los sistemas distribuidos, ya que se refiere a la coherencia de los datos compartidos entre los diferentes nodos del sistema. En un

sistema distribuido, los datos pueden estar replicados en varios nodos para mejorar el rendimiento y la disponibilidad. Sin embargo, esto puede dar lugar a problemas de consistencia, ya que los diferentes nodos pueden tener copias diferentes de los datos [16].

Existen varios modelos de consistencia en sistemas distribuidos, que van desde modelos estrictos que imponen fuertes restricciones de ordenamiento en las operaciones de lectura y escritura, hasta modelos más relajados que permiten cierto grado de reordenamiento de operaciones para mejorar el rendimiento [17].

Los diferentes modelos de consistencia definen diferentes garantías sobre cómo se propagan las actualizaciones de datos en los sistemas distribuidos. Los modelos de consistencia más estrictos, como la consistencia fuerte, garantizan que todos los nodos vean siempre la misma versión de los datos [16]. Los modelos de consistencia más relajados, como la consistencia eventual, permiten que los nodos vean versiones diferentes de los datos durante un período de tiempo limitado [16].

La elección del modelo de consistencia apropiado depende de diversos factores, tales como el tipo de aplicación, la cantidad de nodos en el sistema y la latencia de la red. Los modelos más rigurosos pueden asegurar una mayor consistencia de los datos, pero pueden afectar negativamente el rendimiento, mientras que los modelos más flexibles pueden mejorar el rendimiento, pero permitir cierto grado de inconsistencia en los datos [15].

Uno de los principales retos en la gestión de sistemas distribuidos es mantener la coherencia de la información a lo largo de la red. Es indispensable que los protocolos de coherencia de caché aseguren una visión uniforme y actualizada de los datos, evitando la sobrecarga que introducen los accesos concurrentes sobre un único recurso compartido [15]. Para ello, es crucial que los nombres utilizados por el sistema

sean válidos en toda la red y correspondan con los del hardware, permitiendo así el uso transparente de recursos remotos [18].

La reconfiguración y recuperación ante fallos son acciones que requieren coordinación distribuida, involucrando la información local de todos los componentes del sistema. Los modelos de reconfiguración reportan pocos resultados aplicables a sistemas de control en red y en entornos de tiempo real, lo que destaca la importancia de estudiar estos mecanismos en escenarios con fallas para disminuir su impacto [17]. La simulación de comportamientos de sistemas distribuidos contribuye a obtener robustez y auto-recuperación de fallos, facilitando la tarea de sincronización de datos entre nodos conectados en una red [17], [19].

La escalabilidad de los sistemas distribuidos también es crítica. La capacidad de un sistema para escalar adecuadamente depende de que cada aplicación pueda elegir sus propias abstracciones y reemplazar su implementación según cambie el grado de escalabilidad deseado [20]. Asimismo, la replicación de datos en diferentes nodos y discos duros contribuye a un acceso más rápido a la información y a una mayor resistencia a fallos de componentes individuales, evitando interrupciones del servicio y pérdidas de información [15], [16].

9.2 Coherencia en sistemas distribuidos

La coherencia en sistemas distribuidos se refiere a la propiedad de que las operaciones en ubicaciones compartidas se ejecuten en orden serial. En otras palabras, la coherencia garantiza que las operaciones de lectura y escritura en la misma ubicación de memoria se comporten de manera secuencial y en el orden en que fueron realizadas [16].

La coherencia es un concepto relacionado con la consistencia, pero se refiere a la coherencia de los datos compartidos entre los diferentes procesos que acceden a ellos. En un sistema distribuido, los procesos pueden acceder a los datos compartidos desde diferentes nodos. Sin

embargo, esto puede dar lugar a problemas de coherencia, ya que los diferentes procesos pueden tener diferentes visiones de los datos [19].

Los diferentes modelos de coherencia definen diferentes garantías sobre cómo se propagan las actualizaciones de datos en los sistemas distribuidos. Los modelos de coherencia más estrictos, como la coherencia fuerte, garantizan que todos los procesos vean siempre la misma versión de los datos. Los modelos de coherencia más relajados, como la coherencia eventual, permiten que los procesos vean versiones diferentes de los datos durante un período de tiempo limitado [19].

La coherencia de la información es crítica para mantener la semántica de operación bajo la cual los usuarios escriben sus aplicaciones [20]. Esto implica que los sistemas distribuidos deben garantizar que, pese a la naturaleza descentralizada de su arquitectura, los datos replicados se mantengan actualizados y consistentes en todos los nodos del sistema [21]. El mantenimiento de la coherencia se complica aún más cuando se consideran las posibles fallas de hardware y los errores sistemáticos que pueden ocurrir [22].

Para abordar la coherencia en entornos distribuidos, es común adoptar protocolos de coherencia de caché. Sin embargo, estos protocolos han sido evitados en sistemas de caché paralelos debido a la alta sobrecarga que introducen en los accesos concurrentes sobre un caché [19]. Por lo tanto, se necesitan enfoques alternativos para mantener la coherencia sin comprometer el rendimiento del sistema.

Un elemento crucial en el diseño de sistemas distribuidos es la elección de mecanismos de distribución adecuados. El modelo distribuido impuesto por los Sistemas Operativos Distribuidos convencionales a menudo dificulta o imposibilita que aplicaciones centralizadas aprovechen la distribución y utilicen recursos remotos de manera transparente [21]. Esto sugiere la necesidad de flexibilidad en el

sistema para permitir que cada aplicación elija sus abstracciones dependiendo del grado de escalabilidad deseado [19], [22].

La simulación de comportamientos de sistemas distribuidos juega un papel importante en la experimentación y validación de algoritmos distribuidos y aplicaciones [22]. Estas simulaciones permiten probar la robustez del sistema y su capacidad de auto-recuperación en caso de fallos, lo cual es esencial para mantener la coherencia del sistema [21]. Así, un sistema robusto es capaz de recuperar partes clave del sistema para completar una tarea distribuida de recolección y sincronización de datos [23].

En un escenario ideal, los sistemas distribuidos se basan en una arquitectura cliente-servidor, donde un nodo se configura como servidor y el resto como clientes, con los clientes conociendo la dirección del servidor [24]. Esta configuración permite una coordinación efectiva entre los nodos, facilitando así el mantenimiento de la coherencia.

10. Desafíos de rendimiento en arquitecturas distribuidas

Las arquitecturas distribuidas, que se basan en la interconexión de múltiples componentes de hardware y software en una red, presentan varios desafíos de rendimiento que deben abordarse para garantizar un funcionamiento eficiente y confiable [25]. A continuación, se describen algunos de los desafíos de rendimiento más comunes en arquitecturas distribuidas:

10.1 Latencia de red

La latencia de red es el tiempo que tarda un mensaje o dato en viajar desde un nodo a otro a través de la red. En arquitecturas distribuidas, la latencia de red puede ser un factor limitante para el rendimiento, especialmente en aplicaciones en tiempo real o interacciones sensibles al tiempo. Minimizar la latencia de red es esencial para mejorar el rendimiento [26].

10.2 Ancho de banda

El ancho de banda de la red se refiere a la cantidad de datos que pueden ser transferidos a través de la red en un período de tiempo dado. Si una arquitectura distribuida genera una gran cantidad de tráfico de red, puede sobrecargar la red y reducir el rendimiento. Optimizar el uso del ancho de banda y garantizar un equilibrio adecuado es fundamental[26].

10.3 Consistencia y coherencia de datos

Mantener la coherencia de los datos distribuidos puede ser un desafío. Los protocolos de coherencia de datos deben equilibrar la consistencia con el rendimiento, y la elección del nivel de consistencia adecuado depende de las necesidades de la aplicación [27].

10.4 Distribución de carga

Distribuir la carga de trabajo de manera equitativa entre los nodos distribuidos es esencial para un rendimiento eficiente. Un desequilibrio en la distribución de carga puede provocar la sobrecarga de algunos nodos y la subutilización de otros, lo que afecta negativamente el rendimiento general [27].

10.5 Tolerancia a fallos

Si un componente en una arquitectura distribuida falla, es importante que el sistema pueda recuperarse de manera eficiente sin interrumpir el servicio. Implementar mecanismos de detección y recuperación de fallos, como la redundancia y el failover, es esencial para garantizar un alto rendimiento en presencia de fallos [27].

10.6 Escalabilidad

A medida que una arquitectura distribuida crece y se expande, debe poder escalar para manejar una mayor carga de trabajo. Escalar horizontalmente agregando más nodos a la red es común, pero debe hacerse de manera eficiente para mantener el rendimiento [25].

10.7 Coordinación y sincronización

En entornos distribuidos, la coordinación y la sincronización entre nodos pueden ser complejas y costosas en términos de rendimiento. La elección de algoritmos y técnicas adecuadas para la coordinación y sincronización es crucial para evitar cuellos de botella y retrasos [26].

10.8 Caché distribuido

La gestión de cachés distribuidos puede ser un desafío en arquitecturas distribuidas. Garantizar la coherencia de los datos en caché y la eficiencia en el acceso a los mismos es esencial para el rendimiento [27].

11. Tecnologías emergentes en sistemas distribuidos

Computación en el borde (Edge Computing)

Esta tecnología se ha vuelto esencial con la proliferación de dispositivos IoT y aplicaciones que requieren baja latencia, como los vehículos autónomos y la realidad aumentada. Permite el procesamiento de datos en dispositivos locales, reduciendo la carga en los centros de datos centrales y acelerando las respuestas a eventos en tiempo real. Además, Edge Computing es fundamental para casos de uso que requieren privacidad y seguridad de datos, ya que los datos se mantienen en el borde de la red[28].

11.1 Computación en la nube sin servidor (Serverless)

Las plataformas sin servidor, como AWS Lambda o Azure Functions, permiten a las organizaciones desarrollar aplicaciones sin preocuparse por la infraestructura subyacente. Esto conduce a una mayor eficiencia en el desarrollo y la escalabilidad automática según la demanda. Los desarrolladores solo se centran en el código y pagan por el tiempo real de ejecución, lo que puede resultar en costos más bajos y un tiempo de desarrollo más rápido [28].

11.2 Blockchain y tecnologías de registro distribuido (DLT)

Estas tecnologías están revolucionando la forma en que se gestionan los registros y las transacciones en sistemas distribuidos. Blockchain es conocido por su uso en criptomonedas como Bitcoin, pero también se aplica en la gestión de cadenas de suministro, votaciones electrónicas, contratos inteligentes y más. Proporciona una capa de seguridad y confiabilidad que es crucial en aplicaciones críticas [29].

11.3 Microservicios y contenedores

La arquitectura de microservicios se ha convertido en un enfoque popular para el desarrollo de aplicaciones distribuidas. Los contenedores, como Docker, permiten empacar y desplegar microservicios de manera eficiente y aislada. Esto facilita la escalabilidad y la administración de aplicaciones complejas, ya que cada microservicio se puede desarrollar, probar y actualizar de forma independiente [29].

11.4 Orquestación de contenedores

Kubernetes es una de las plataformas de orquestación de contenedores más utilizadas. Facilita la administración de clústeres de contenedores y proporciona escalabilidad, alta disponibilidad y autoreparación. La orquestación de contenedores es fundamental para implementaciones eficientes y resilientes en sistemas distribuidos [30].

11.5 Procesamiento de streaming y análisis en tiempo real

Con la creciente cantidad de datos generados en tiempo real, el procesamiento de streaming se ha vuelto crucial. Tecnologías como Apache Kafka permiten la ingestión, el procesamiento y la entrega de flujos de datos en tiempo real. Esto es fundamental para aplicaciones de análisis en tiempo real, como la detección de anomalías y la personalización de contenido [30].

11.6 Inteligencia Artificial (IA) distribuida

La IA y el aprendizaje automático se utilizan en sistemas distribuidos para mejorar la toma de decisiones y automatizar tareas. La distribución de tareas de IA en múltiples nodos permite un procesamiento más rápido y eficiente de grandes conjuntos de datos, lo que es esencial en aplicaciones como el análisis de big data y la visión por computadora[28].

11.7 Redes 5G

El despliegue de redes 5G ofrece velocidades de conexión más rápidas y menor latencia. Esto es fundamental para aplicaciones distribuidas que requieren una conectividad confiable y de alto rendimiento, como vehículos autónomos, telemedicina y realidad virtual/aumentada [29].

11.8 Seguridad distribuida

La seguridad en sistemas distribuidos es crítica, y las tecnologías emergentes se centran en la autenticación, autorización y cifrado distribuido. Además, se utilizan enclaves seguros y tecnologías de confidencialidad para proteger los datos sensibles en sistemas distribuidos [25].

11.9 Arquitecturas de nube híbrida y multi-nube

La flexibilidad y la redundancia se logran mediante la adopción de arquitecturas de nube híbrida y multi-nube. Esto permite a las organizaciones equilibrar costos y rendimiento al distribuir sus aplicaciones y servicios en múltiples entornos de nube pública y privada [30].

12. Integración de contenedores en arquitecturas distribuidas

Los contenedores se pueden integrar en arquitecturas distribuidas mediante el uso de tecnologías de contenedores para una implementación ligera de aplicaciones [31]. Los contenedores proporcionan potentes mecanismos de aislamiento, lo que permite a

los desarrolladores centrarse en la lógica de las aplicaciones mientras los administradores del sistema se ocupan de la implementación y la administración [32]. Los sistemas basados en contenedores se pueden modelar mediante sistemas bigráficos reactivos (BRS), lo que permite el análisis y la manipulación mediante técnicas de teoría de grafos [33]. Están surgiendo patrones de diseño para los sistemas distribuidos basados en contenedores, incluidos patrones para la administración de contenedores, la estrecha cooperación de los contenedores en un solo nodo y los algoritmos distribuidos en varios nodos [32]. Los contenedores también ofrecen una solución para ejecutar y evaluar aplicaciones de investigación en conjuntos de datos sensibles o de gran tamaño, lo que permite a los investigadores obtener información sin necesidad de transferir los datos [33]. Los métodos automatizados de toma de decisiones, como los procesos estocásticos de decisión de Markov (MDP), se pueden utilizar para optimizar la ubicación de los contenedores en arquitecturas distribuidas y garantizar una alta calidad de servicio (QoS) . Kubernetes se puede usar para organizar el despliegue de contenedores en infraestructuras periféricas, de niebla y de nube [31].

La integración de contenedores en arquitecturas distribuidas es una práctica esencial en el desarrollo de aplicaciones modernas. Los contenedores son entornos ligeros y portátiles que permiten empaquetar aplicaciones y sus dependencias en un formato único, lo que facilita su implementación, escalabilidad y gestión en sistemas distribuidos [28].

12.1 Empaquetado de aplicaciones

Los contenedores permiten empaquetar aplicaciones y todas sus dependencias en una unidad independiente. Esto incluye bibliotecas, archivos de configuración y cualquier otro recurso necesario. Como resultado, las aplicaciones se ejecutan de manera consistente en

diferentes entornos, lo que simplifica el proceso de implementación [32].

12.2 Portabilidad

Los contenedores son portátiles, lo que significa que pueden ejecutarse en cualquier entorno de contenedor compatible, independientemente del sistema operativo o la infraestructura subyacente. Esto facilita la migración de aplicaciones entre entornos locales, en la nube o en diferentes proveedores de servicios en la nube [31].

12.3 Aislamiento

Cada contenedor se ejecuta en un entorno aislado, lo que significa que no comparte recursos ni dependencias con otros contenedores en el mismo sistema. Esto garantiza que las aplicaciones no entren en conflicto entre sí y mejora la seguridad [33].

12.4 Escalabilidad

Los contenedores son ideales para implementaciones escalables. Puedes replicar contenedores para manejar cargas de trabajo crecientes y utilizar herramientas de orquestación, como Kubernetes, para administrar automáticamente el escalado y la distribución de contenedores en entornos distribuidos [32].

12.5 Facilita la gestión

Los contenedores simplifican la gestión de aplicaciones distribuidas. Puedes automatizar la implementación, actualización y monitorización de contenedores, lo que facilita la administración de aplicaciones complejas en entornos distribuidos [34].

12.6 Control de versiones y reproducibilidad

Los contenedores permiten un control de versiones más efectivo de las aplicaciones y sus dependencias. Puedes definir la configuración de contenedor en código (infraestructura como código) para garantizar la reproducibilidad de las implementaciones [33].

12.7 Entornos de desarrollo consistentes

Los desarrolladores pueden trabajar en entornos de desarrollo locales que replican con precisión los entornos de producción basados en contenedores. Esto evita problemas de "funciona en mi máquina" y agiliza el ciclo de desarrollo [33].

12.8 Integración con herramientas de CI/CD

Los contenedores se integran perfectamente en los flujos de trabajo de Integración Continua y Entrega Continua (CI/CD). Esto permite automatizar la construcción, prueba y despliegue de aplicaciones de manera eficiente en arquitecturas distribuidas [32].

12.9 Facilita la adopción de microservicios

Los contenedores son ideales para implementar microservicios, ya que permiten que cada servicio se ejecute en su propio contenedor. Esto facilita la gestión y la escalabilidad de microservicios en arquitecturas distribuidas [32].

13. Arquitectura microservicios vs monolíticas

13.1 Microservicios

La arquitectura de microservicios fomenta el desarrollo y la implementación de aplicaciones compuestas por unidades autónomas, modulares y autocontenidas, en contraste con el enfoque tradicional o monolítico [26].

Los microservicios emergen con el propósito de superar los problemas que tenían los sistemas monolíticos, que mantienes los servicios como unidad lógica usando los mismos recursos de computadoras. Muchos sistemas de información en el mundo están hechos de esta manera, y debido a las limitaciones que impone para mejorar, las organizaciones están buscando cambiar a un enfoque más moderno, aprovechando las infraestructuras en la nube.

13.1.1 Características

En una arquitectura de microservicios, los servicios son pequeños e independientes, con un acoplamiento flexible. Cada servicio representa un código base autónomo, gestionado por un equipo de desarrollo pequeño [27].

Los servicios pueden implementarse de forma independiente, permitiendo actualizar sin regenerar la aplicación. Cada servicio es responsable de conservar sus propios datos o estado externo, a diferencia del modelo tradicional.

La comunicación entre servicios se realiza mediante API bien definidas, ocultando la implementación interna de cada servicio.

No es necesario que los servicios compartan la misma pila tecnológica, bibliotecas o marcos de trabajo [26].

13.2 Monolíticos

Un monolito se define como un sistema en el que todos los módulos pertinentes están agrupados en una única entidad, y donde todas las funcionalidades deben desplegarse conjuntamente. En este tipo de arquitectura, la vista, la lógica de negocio y la capa de acceso a los datos son componentes que coexisten en un único sistema [27].

13.2.1 Características

El código creado para abarcar los procesos de todo el sistema se guarda en una única base de datos. Aunque existe la opción de dividir el código en diversas clases y paquetes para lograr una estructura más organizada, los módulos permanecen sin división, actuando como una única entidad [27].

Si se necesita efectuar algún cambio en la aplicación, es imprescindible analizar y modificar la totalidad del software. La integración se debe realizar en partes pequeñas, ya que las secciones no pueden operar de forma independiente; se requiere una integración completa en el sistema [27].

Los componentes dentro de la aplicación pueden comunicarse fácilmente mediante la utilización de procedimientos internos [27].

13.3 Automatización en implementaciones distribuidas

Al automatizar tareas como la configuración inicial, la orquestación de componentes y la gestión de recursos, se busca mejorar la eficiencia y reducir errores humanos en entornos distribuidos. La automatización también facilita la implementación rápida de cambios y actualizaciones, contribuyendo a la agilidad del sistema [28].

13.4 Herramientas de implementación continua (CI/CD)

El objetivo práctico de la integración continua es establecer una manera consistente y automatizada de crear, empaquetar y probar aplicaciones. Al mantener un proceso constante de integración, es más probable que los equipos hagan cambios en el código con regularidad, promoviendo una mejor colaboración y calidad en el software [28].

13.5 Infraestructura como código

La idea de "infraestructura como código" (IaC) consiste en manejar los sistemas y dispositivos usados para correr software como si fueran software en sí. Es crucial para la práctica de DevOps, especialmente en la entrega constante de software. Asegurarse de que los scripts de IaC estén libres de problemas de seguridad es fundamental para evitar introducir debilidades de seguridad en la infraestructura tecnológica o en los sistemas que se gestionan [29].

14. Desarrollo de aplicaciones escalables en entornos distribuidos

La escalabilidad en sistemas distribuidos es un componente esencial para garantizar que un sistema pueda adaptarse eficazmente al aumento de la carga de trabajo o al crecimiento del número de usuarios. La escalabilidad se define como la capacidad del sistema para

manejar el aumento de la demanda sin comprometer su rendimiento[30].

Existen dos enfoques principales para lograr escalabilidad: la escalabilidad horizontal, que implica agregar más nodos o recursos, y la escalabilidad vertical, que implica mejorar los recursos individuales de un nodo [3].

Sin embargo, la escalabilidad en sistemas distribuidos presenta desafíos específicos. La gestión de la coherencia de datos distribuidos y la coordinación entre nodos son áreas críticas que pueden afectar el rendimiento. Los cuellos de botella también pueden surgir por la complejidad de la distribución de tareas entre diferentes componentes [31].

Para abordar estos desafíos, se recurre a arquitecturas diseñadas específicamente para la escalabilidad, como la arquitectura de microservicios y la arquitectura basada en eventos. Estas arquitecturas permiten un crecimiento modular al dividir la aplicación en servicios más pequeños y especializados [31].

El particionamiento de datos y el balanceo de carga son estrategias fundamentales en la búsqueda de una distribución equitativa de la carga entre los nodos. El particionamiento de datos contribuye a la distribución eficiente, mientras que el balanceo de carga asegura que las solicitudes se distribuyan de Calibri manera equitativa, optimizando así el rendimiento del sistema [31].

14.1 Impacto en la nube de las arquitecturas distribuidas

Las arquitecturas distribuidas han transformado significativamente el panorama en la nube, generando un impacto profundo en la forma en que las aplicaciones se diseñan y gestionan en entornos de computación en la nube. Un aspecto clave es la escalabilidad horizontal inherente a las arquitecturas distribuidas, que se alinea perfectamente con la naturaleza elástica de la nube [32].

Esta capacidad permite a las aplicaciones distribuidas adaptarse dinámicamente a variaciones en la carga de trabajo mediante la adición de recursos de manera eficiente [33].

Las arquitecturas basadas en microservicios, otra característica distintiva de las implementaciones distribuidas se ha convertido en un estándar en la nube. La capacidad de dividir aplicaciones en servicios independientes permite un enfoque modular que simplifica el despliegue y la escalabilidad, aspectos críticos en entornos de nube dinámicos [33].

El auge de contenedores, como Docker, y herramientas de orquestación, como Kubernetes, ha sido un componente esencial en la evolución de arquitecturas distribuidas en la nube. Estas tecnologías facilitan la portabilidad de las aplicaciones distribuidas y permiten su despliegue consistente en diferentes entornos en la nube, contribuyendo a la gestión eficiente de recursos [33].

15. Practica

15.1 Problema

Ineficiencia en la Gestión de Datos de Usuarios en Aplicaciones Web

15.2 Descripción

La problemática radica en la ineficiencia en la gestión de datos de usuarios en aplicaciones web, incluyendo la falta de una interfaz de usuario intuitiva y segura, así como la gestión adecuada de operaciones CRUD en un servidor web. Para resolver esto, hemos desarrollado una solución que combina Flask y PostgreSQL. Creamos una interfaz de usuario amigable que permite a los usuarios agregar, editar y eliminar registros de usuarios de manera sencilla. Utilizamos PostgreSQL para una gestión eficiente y segura de la base de datos, implementando medidas de seguridad como la validación de entradas y el almacenamiento seguro de contraseñas. La estructura de código

modular y las buenas prácticas de codificación garantizan su mantenimiento y escalabilidad, ofreciendo una solución moderna y eficaz para la gestión de datos de usuarios en aplicaciones web.

15.3 Código paso a paso

1. Establecer la conexión con la base de datos

En el archivo database.py, se establece la conexión con la base de datos PostgreSQL.

```
import psycopg2

# Estableciendo la conexión
database = psycopg2.connect(
    host='localhost',
    user='postgres',
    password='12345',
    dbname='ArquiDisti'
)
```

Ilustración 1: Fragmento de código para conexión con la base de datos PostgreSQL

Este código utiliza la librería psycopg2 para conectarse a la base de datos PostgreSQL. Nos debemos de asegurar que los campos sean rellenados con la información de nuestra base de datos.

2. Configuración de la Aplicación Flask

El archivo app.py configura la aplicación Flask y define las rutas.

```
from flask import Flask, redirect, render_template, request, url_for
import os
import database as db

template_dir=os.path.dirname(os.path.abspath(os.path.dirname(__file__)))
template_dir=os.path.join(template_dir, 'src', 'templates')
app = Flask(__name__, template_folder = template_dir)
```

Ilustración 2: Fragmento de código para configuración del framework Flask

Este código configura la aplicación Flask y establece la ubicación de las plantillas HTML.

- Importamos las bibliotecas necesarias de Flask y otros módulos.
- Configuramos la ubicación de la carpeta de plantillas HTML, asegurándonos de que esté correctamente configurada según la estructura de archivos.

3. Desarrollamos la ruta principal (home)

```
# Rutas de la app
@app.route('/')
def home():

    cursor=db.database.cursor()
    cursor.execute('SELECT * FROM "usuarios"')
    myresult = cursor.fetchall()

    insertObject = []
    columnNames = [column[0] for column in cursor.description]
    for record in myresult:
        insertObject.append(dict(zip(columnNames, record)))
    cursor.close()

    return render_template('index.html', data=insertObject)
```

Ilustración 3: Fragmento de código que maneja la ruta principal

Este método se encarga de manejar la ruta principal **'/'**, que muestra la lista de usuarios desde la base de datos en la página principal de la aplicación. Aquí está lo que hace cada parte del método:

- Se crea un objeto cursor para interactuar con la base de datos.
- Se ejecuta una consulta SQL para seleccionar todos los registros de la tabla **"usuarios"**.
- Los resultados de la consulta se almacenan en **myresult**.
- Se crea una lista **insertObject** para almacenar los datos de los usuarios en un formato más manejable.
- Se obtienen los nombres de las columnas de la tabla y se almacenan en **columnNames**.

- Se itera a través de los resultados de la consulta y se crea un diccionario para cada registro, donde las claves son los nombres de las columnas y los valores son los datos de los usuarios.
- Se cierra el cursor.
- Finalmente, se renderiza la plantilla HTML **'index.html'** y se pasa la lista **insertObject** como datos para mostrar en la página.

4. Desarrollamos la ruta para aregar (adduser)

```python
@app.route('/user', methods=['POST'])
def addUser():
    username = request.form['username']
    name = request.form['name']
    password = request.form['password']

    if username and name and password:
        cursor = db.database.cursor()
        sql = "INSERT INTO usuarios (username, name, password) VALUES (%s, %s, %s)"
        data = (username, name, password)
        cursor.execute(sql, data)
        db.database.commit()
    return redirect(url_for('home'))
```

Ilustración 4: Fragmento de código que permite registrar nuevos usuarios

Este método maneja la ruta **/user**, que recibe solicitudes POST para agregar nuevos usuarios a la base de datos. Aquí está lo que hace cada parte del método:

- Se obtienen los datos del formulario enviado mediante la solicitud POST, incluyendo **username**, **name** y **password**.
- Se verifica que los datos recibidos no estén vacíos.
- Se crea un cursor para interactuar con la base de datos.
- Se define una consulta SQL para insertar un nuevo usuario en la tabla **"usuarios"**.
- Se crea una tupla **data** que contiene los valores a insertar en la consulta.
- Se ejecuta la consulta SQL y se confirma la transacción en la base de datos.

- Luego, se redirige de nuevo a la página principal.

5. Desarrollamos la ruta para eliminar (Delete)

```
@app.route('/delete/<string:id>')
def delete(id):
    cursor = db.database.cursor()
    sql = "DELETE FROM usuarios WHERE id=%s"
    data=(id)
    cursor.execute(sql,data)
    db.database.commit()
    return redirect(url_for('home'))
```

Ilustración 5: Fragmento de código que permite eliminar un usuario registrado

Este método maneja la ruta **/delete/<string:id>**, que recibe solicitudes para eliminar usuarios por su ID. Aquí está lo que hace cada parte del método:

- Se crea un cursor para interactuar con la base de datos.
- Se define una consulta SQL para eliminar un usuario de la tabla **"usuarios"** basado en su ID.
- Se crea una tupla **data** que contiene el valor del ID a eliminar.
- Se ejecuta la consulta SQL y se confirma la transacción en la base de datos.
- Luego, se redirige de nuevo a la página principal.

6. Desarrollamos la ruta para editar (Edit)

```python
@app.route('/edit/<string:id>', methods=['POST'])
def edit(id):
    username = request.form['username']
    name = request.form['name']
    password = request.form['password']

    if username and name and password:
        cursor = db.database.cursor()
        sql = "UPDATE usuarios SET username = %s, name = %s, password = %s WHERE id = %s"
        data = (username, name, password, id)
        cursor.execute(sql, data)
        db.database.commit()
        return redirect(url_for('home'))
```

Ilustración 6: Fragmento de código que permite editar un usuario ya registrado

Este método maneja la ruta **/edit/<string:id>**, que recibe solicitudes POST para editar la información de usuarios existentes. Aquí está lo que hace cada parte del método:

- Se obtienen los datos del formulario enviado mediante la solicitud POST, incluyendo **username**, **name** y **password**.
- Se verifica que los datos recibidos no estén vacíos.
- Se crea un cursor para interactuar con la base de datos.
- Se define una consulta SQL para actualizar los datos de un usuario en la tabla **"usuarios"** basado en su ID.
- Se crea una tupla **data** que contiene los nuevos valores para el usuario.
- Se ejecuta la consulta SQL y se confirma la transacción en la base de datos.
- Luego, se redirige de nuevo a la página principal.

Nota

Estos son los métodos clave que manejan las operaciones de la aplicación Flask relacionadas con la base de datos y la interfaz de usuario. Cada uno realiza tareas específicas para agregar, eliminar y editar usuarios en la base de datos.

7. Desarrollamos el interfaz gráfica (index.html)

Head

```
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-Zen
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/js/bootstrap.bundle.min.js" integrity="sha384-OERcA2EqjJCMA+
    <style>
        body {
            background-color: #f8f9fa;
        }
        .card {
            border-radius: 15px;
        }
        .modal-content {
            border-radius: 15px;
        }
        .btn {
            border-radius: 5px;
        }
    </style>
</head>
```

Ilustración 7: Fragmento de código html para el encabezado

Body (AddUser)

```
<div class="container py-5">
    <h1 class="text-center mb-5 text-primary">Python-Flask-PostgreSQL App</h1>

    <div class="card shadow-sm p-4">
        <form action="/user" method="POST">
            <div class="row g-3">
                <div class="col-md-3">
                    <label for="username" class="form-label">Correo</label>
                    <input type="text" class="form-control" name="username" id="username">
                </div>
                <div class="col-md-3">
                    <label for="name" class="form-label">Nombre</label>
                    <input type="text" class="form-control" name="name" id="name">
                </div>
                <div class="col-md-3">
                    <label for="password" class="form-label">Contraseña</label>
                    <input type="password" class="form-control" name="password" id="password">
                </div>
                <div class="col-md-3 d-flex align-items-end">
                    <button class="btn btn-primary w-100" type="submit">Guardar</button>
                </div>
            </div>
        </form>
```

Ilustración 8: Fragmento de código html para registro de usuarios

Body (Tabla de registro)

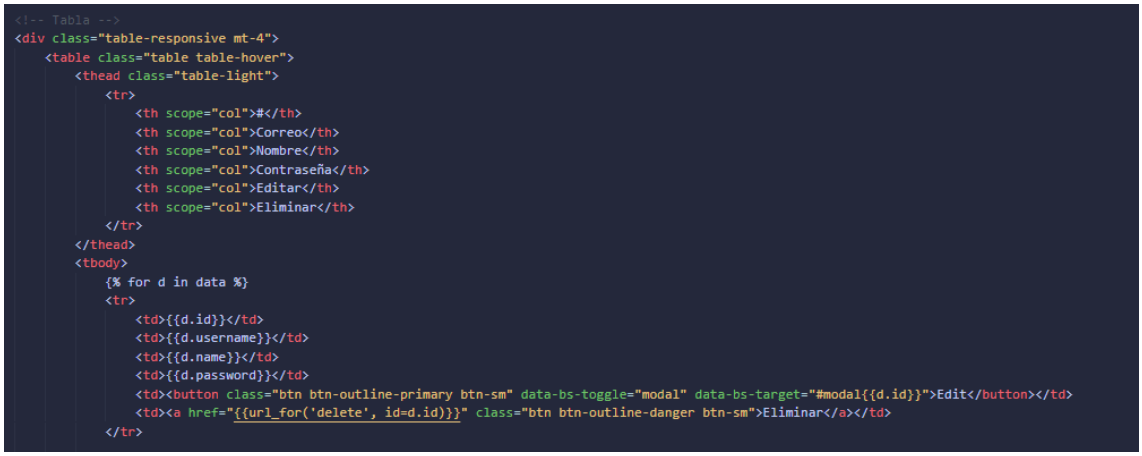

Ilustración 9: Fragmento de código html para visualización de información

Body (Modal)

Ilustración 10: Fragmento de código html para editar un usuario ya registrado

- **Encabezado (Header)**:
  - Se define la codificación y el título de la página.
  - Se importan las hojas de estilo Bootstrap y los scripts de Bootstrap para el funcionamiento de componentes interactivos.
- **Cuerpo (Body)**:

    - Se establece el fondo de la página con el color **#f8f9fa**.
    - Se crea un contenedor principal con la clase **container py-5** que contiene todo el contenido de la página.
- **Título**:
    - Se agrega un título principal centrado en la página con el texto "Python-Flask-PostgreSQL App".
- **Formulario para Agregar Usuarios**:
    - Se crea un formulario que permite agregar nuevos usuarios a la base de datos.
    - El formulario tiene campos para **Correo**, **Nombre**, y **Contraseña**.
    - Los campos de entrada tienen etiquetas (labels) que describen su propósito.
    - Se utiliza la clase **form-control** de Bootstrap para estilizar los campos de entrada.
    - Se agrega un botón "Guardar" para enviar el formulario.
- **Tabla de Usuarios**:
    - Se crea una tabla que muestra la lista de usuarios recuperada de la base de datos.
    - La tabla tiene encabezados de columna con las etiquetas: **#**, **Correo**, **Nombre**, **Contraseña**, **Editar**, y **Eliminar**.
    - Se utiliza la clase **table** de Bootstrap para dar estilo a la tabla.
    - Se utiliza la clase **table-light** para dar estilo a la fila de encabezados.
- **Iteración a través de los Usuarios**:

- Se utiliza una estructura de bucle para iterar a través de los datos de los usuarios obtenidos de la base de datos.
- Por cada usuario, se crea una fila en la tabla que muestra su información en las columnas correspondientes.
- Para cada usuario, se agrega un botón "Editar" que permite editar sus datos y un botón "Eliminar" que permite eliminarlo.
- Se utiliza Bootstrap para dar estilo a los botones y para mostrar un modal de edición al hacer clic en "Editar".

- **Modal de Edición**:
  - Se crea un modal de Bootstrap para la edición de usuarios.
  - El modal muestra los campos **Correo**, **Nombre**, y **Contraseña** del usuario seleccionado.
  - Al hacer cambios y hacer clic en "Guardar cambios", se envían los datos al servidor para actualizar el usuario.

15.4 Interfaces

Principal sin registros

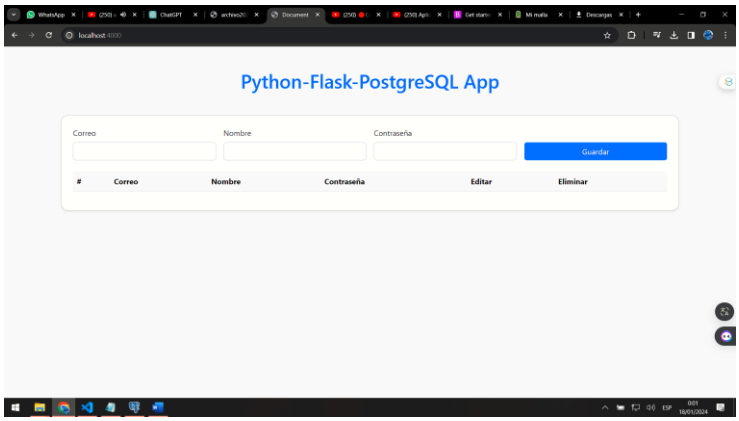

Ilustración 11: Interfaz principal sin datos agregados

Principal con registros

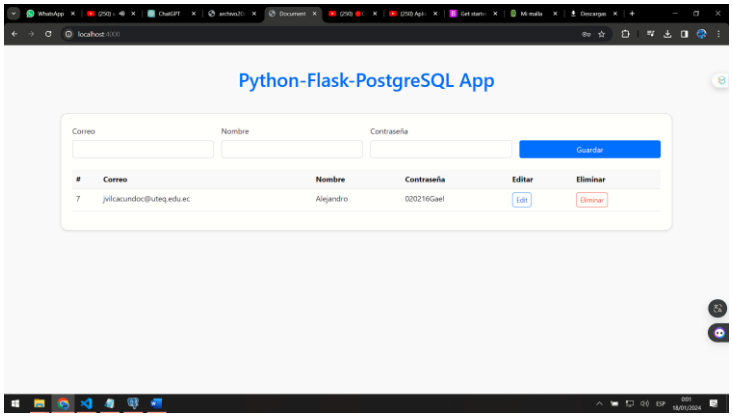

Ilustración 12: Interfaz principal con datos agregados

Modal de edición

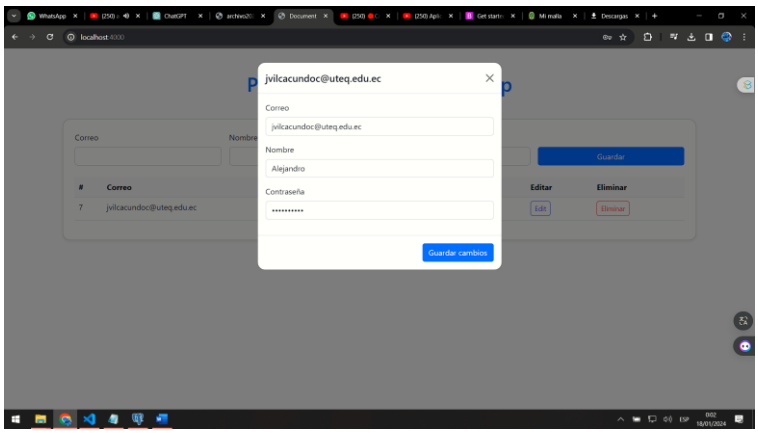

Ilustración 13: Interfaz del modal para edición de registros

16. Conclusión

En resumen, las arquitecturas distribuidas son fundamentales para el desarrollo moderno de sistemas, ofreciendo ventajas como redundancia y tolerancia a fallos. Sin embargo, enfrentan desafíos como la latencia y la consistencia de datos.

La adopción de tecnologías emergentes como la computación sin servidor, blockchain y microservicios está transformando la forma en que construimos aplicaciones distribuidas. Estrategias de despliegue y la gestión eficiente de desafíos de rendimiento son cruciales para garantizar un funcionamiento óptimo.

La integración de contenedores, la automatización en implementaciones distribuidas y la elección entre arquitecturas microservicios y monolíticas son decisiones clave. La comprensión del impacto en la nube y la implementación efectiva de prácticas como la infraestructura como código son esenciales en este entorno dinámico y tecnológico.

El desarrollo de aplicaciones escalables en entornos distribuidos es esencial para adaptarse a la naturaleza dinámica de la nube. La comprensión del impacto de las arquitecturas distribuidas en la nube y la implementación efectiva de prácticas como la infraestructura como código son clave para garantizar el éxito en un mundo cada vez más distribuido y orientado a la tecnología.

Por otro lado, la gestión de tareas con microservicios en Python, Flask y PostgreSQL proporciona una solución ágil y escalable. La arquitectura modular de microservicios facilita el desarrollo, despliegue y mantenimiento eficiente. La elección de Python y Flask agrega simplicidad, mientras que PostgreSQL asegura robustez en la gestión de datos. La implementación de DevOps y prácticas de CI/CD optimiza el ciclo de vida de desarrollo, aunque se deben abordar desafíos de coordinación entre microservicios y consistencia de datos distribuidos. En conjunto, esta solución moderna permite adaptarse ágilmente a las demandas empresariales cambiantes.

17. Link del Github

https://github.com/OrlandCede20/PRACT_ARQ_DISTR.git

18. Referencias

[1] D. Casas Lizcano and F. de A. López, *Sistemas distribuidos*, no. 1. 2015. [Online]. Available: http://www1.frm.utn.edu.ar/soperativos/Archivos/Sistemas_Distribuidos.pdf

[2] S. Inform, “DISTRIBUTED SYSTEMS,” pp. 1–144, 2003.

[3] Eduardo Andrade and Gonzalo Muñoz, “Sistemas Distribuidos, Escalabilidad Y Distribución De Los Datos,” 2010.

[4] M. van Steen and A. S. Tanenbaum, *Distributed Systems Third edition*, vol. 28, no. 11. 1985.

[5] A. M. G. · T. A. Hrsg, *Lexikon der Medizinischen Laboratoriums diagnostik*. 2019. [Online]. Available: www.springerreference.de.

[6] S. Zhang, Q. Wang, and Y. Kanemas, “Improving asynchronous invocation performance in client-server systems,” in *Proceedings - International Conference on Distributed Computing Systems*, 2018. doi: 10.1109/ICDCS.2018.00092.

[7] J. Cao, D. F. Bucher, D. M. Hall, and J. Lessing, “Cross-phase product configurator for modular buildings using kit-of-parts,” *Autom. Constr.*, vol. 123, 2021, doi: 10.1016/j.autcon.2020.103437.

[8] T. W. Cheng, W. L. Wang, and S. K. Chan, “Three-tier multi-agent architecture for asset management consultant,” in *Proceedings - 2004 IEEE International Conference on e-Technology, e-Commerce and e-Service, EEE 2004*, 2004. doi: 10.1109/eee.2004.1287305.

[9] M. Kim, Q. Xin, S. Sinha, and A. Orso, “Automated test generation for REST APIs: No time to rest yet,” in *ISSTA 2022 - Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022. doi: 10.1145/3533767.3534401.

[10] A. Neumann, N. Laranjeiro, and J. Bernardino, “An Analysis of Public REST Web Service APIs,” *IEEE Trans. Serv. Comput.*, vol. 14, no. 4, 2021, doi: 10.1109/TSC.2018.2847344.

[11] A. Soni and V. Ranga, “API features individualizing of web services: REST and SOAP,” *Int. J. Innov. Technol. Explor. Eng.*,

vol. 8, no. 9 Special Issue, 2019, doi: 10.35940/ijitee.I1107.0789S19.

[12] A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, and D. Mishra, "RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions," *Applied Sciences (Switzerland)*, vol. 12, no. 9. 2022. doi: 10.3390/app12094369.

[13] S. B. Cleveland *et al.*, "Tapis API Development with Python: Best Practices in Scientific REST API Implementation: Experience implementing a distributed Stream API," in *ACM International Conference Proceeding Series*, 2020. doi: 10.1145/3311790.3396647.

[14] A. Luntovskyy and J. Spillner, *Architectural Transformations in Network Services and Distributed Systems*. 2017. doi: 10.1007/978-3-658-14842-3.

[15] R. Topor, "Safety and Domain Independence," in *Encyclopedia of Database Systems*, 2016. doi: 10.1007/978-1-4899-7993-3_1255-2.

[16] A. W. Mohamed and A. M. Zeki, "Web services SOAP optimization techniques," in *4th IEEE International Conference on Engineering Technologies and Applied Sciences, ICETAS 2017*, 2017. doi: 10.1109/ICETAS.2017.8277881.

[17] M. M. Alshammari, "Evolution of Issues in Distributed Systems: A Systematic Review," *Proc. - 2020 19th Distrib. Comput. Appl. Bus. Eng. Sci. DCABES 2020*, pp. 33–37, 2020, doi: 10.1109/DCABES50732.2020.00018.

[18] Z. Li, G. Xie, and Z. Li, "Efficient and scalable consistency maintenance for heterogeneous peer-to-peer systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 12, pp. 1695–1708, 2008, doi: 10.1109/TPDS.2008.46.

[19] X. Wang, B. Shi, and Y. Fang, "Distributed Systems for Emerging Computing: Platform and Application," *Future Internet*, vol. 15, no. 4. MDPI, Apr. 01, 2023. doi: 10.3390/fi15040151.

[20] N. Cai, S. Wei, F. Wang, H. Deng, B. Liang, and W. Dai, "A survey on stream distributed computing," *Proc. - 8th Int. Conf. Intell. Networks Intell. Syst. ICINIS 2015*, pp. 94–97, 2016, doi: 10.1109/ICINIS.2015.44.

[21] R. Sahl, P. Dupont, C. Messager, M. Honnorat, and T. Vu La, "High-Resolution Ocean Winds: Hybrid-Cloud Infrastructure for Satellite Imagery Processing," *IEEE Int. Conf. Cloud Comput. CLOUD*, vol. 2018-July, pp. 883–886, 2018, doi: 10.1109/CLOUD.2018.00127.

[22] D. Wei, S. Basu Roy, and S. Amer-Yahia, "Recommending Deployment Strategies for Collaborative Tasks," *Proc. ACM SIGMOD Int. Conf. Manag. Data*, pp. 3–17, 2020, doi: 10.1145/3318464.3389719.

[23] W. Wong, A. Zavodovski, P. Zhou, and J. Kangasharju, "Container deployment strategy for edge networking," *MECC 2019 - Proc. 2019 4th Work. Middlew. Edge Clouds Cloudlets, Part Middlew. 2019*, pp. 1–6, 2019, doi: 10.1145/3366614.3368101.

[24] Y. Xie, K. Wu, T. Zhao, and X. Gao, "Research and Design of Dynamic Strategy Distributed Control Architecture in Power Internet of Things," *Proc. - 2020 Int. Conf. Artif. Intell. Comput. Eng. ICAICE 2020*, pp. 403–407, 2020, doi: 10.1109/ICAICE51518.2020.00085.

[25] C. H. Lee, J. Kwak, and D. Y. Eun, "Towards Distributed Optimal Movement Strategy for Data Gathering in Wireless Sensor Networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 2, pp. 574–584, 2016, doi: 10.1109/TPDS.2015.2407893.

[26] W. E. B. D. E. La and A. Nacional, "ARQUITECTURA DE SOFTWARE

BASADA EN MICROSERVICIOS PARA DESARROLLO DE APLICACIONES WEB DE LA ASAMBLEA NACIONA," 2017.

[27] D. Maya, E., y López, "Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web," *Septima Conf. Dir. Tecnol. Inf. 2017*, no. July, p. 12, 2017, [Online]. Available: http://dspace.redclara.net:8080/bitstream/10786/1277/1/93 Arquitectura de Software basada en Microservicios para Desarrollo de Aplicaciones Web.pdf

[28] G. J. Caiza and M. V. García, "Implementación de sistemas distribuidos de bajo costo bajo norma IEC-61499, en la estación de clasificación y manipulación del MPS 500," *Ingenius*, no. 18, p. 40, 2017, doi: 10.17163/ings.n18.2017.05.

[29] J. Moreno Martínez, "CI/CD en Infraestructura como código (IaC). Caso práctico en Amazon Web Services (AWS).," 2022.

[30] F. D. Muñoz Escoí *et al.*, *Concurrencia y sistemas distribuidos*. 2012.

[31] M. R. P. Lozoya, "Interconexión de sistemas distribuidos," *Universi*, 2010.

[32] T. F. De Máster, "Arquitectura Distribuida en la Nube para el cálculo específico de software mediante," 2019.

[33] A. R. Anggraini and J. Oliver, "Diseño E Implementación De Sistema Distribuido Y Colaborativo De Peticiones Http/S," *J. Chem. Inf. Model.*, vol. 53, no. 9, p. 7, 2019, [Online]. Available: http://repositorio.uchile.cl/bitstream/handle/2250/168613/Diseño-e-implementación-de-sistema-distribuido-y-colaborativo-de.pdf?sequence=1&isAllowed=y