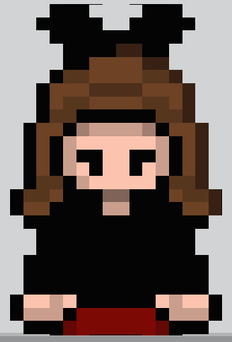
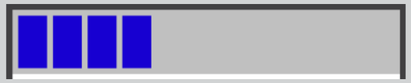
SISTEMAS DISTRIBUIDOS

ZMQ

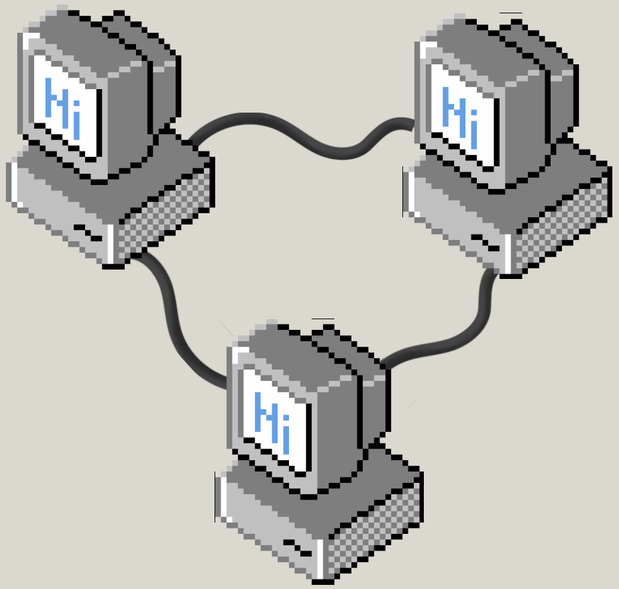




Sofia Cespedes Vargas

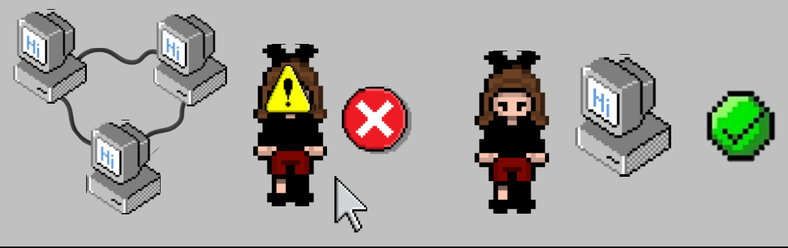
SISTEMAS DISTRIBUIDOS

Es un conjunto de componentes informáticos que se ejecutan en diferentes nodos (computadoras, servidores, etc.) pero que trabajan juntos como si fueran una única entidad, coordinados para lograr un objetivo común.



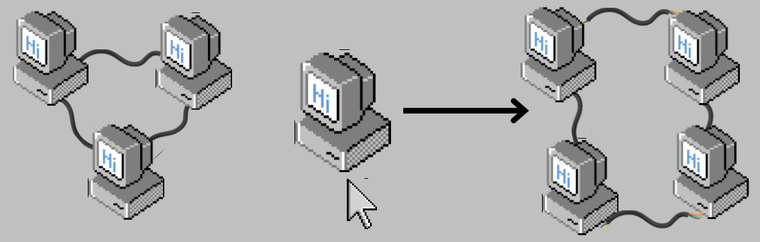
CARACTERISTICAS

Transparencia



El sistema debe ocultar a los usuarios el hecho de que sus procesos y recursos están distribuidos

Escalabilidad

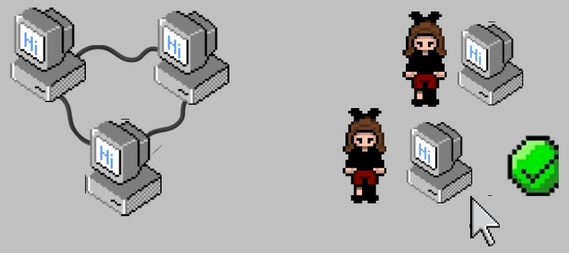


La arquitectura del sistema debe permitir que se añadan más nodos (máquinas) sin una degradación significativa del rendimiento.

Tolerancia a fallos

El sistema debe poder recuperarse de fallos parciales, sin comprometer la disponibilidad global.

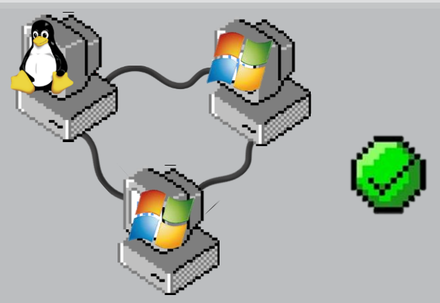
Concurrencia



Varios procesos o usuarios pueden interactuar simultáneamente con los recursos distribuidos sin causar inconsistencias ni errores.

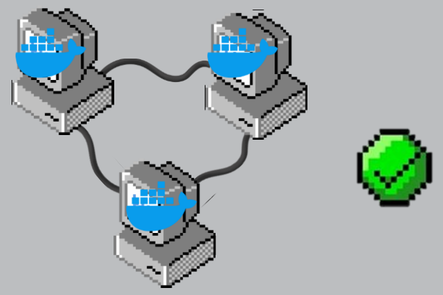
CARACTERISTICAS

Hetereogenidad



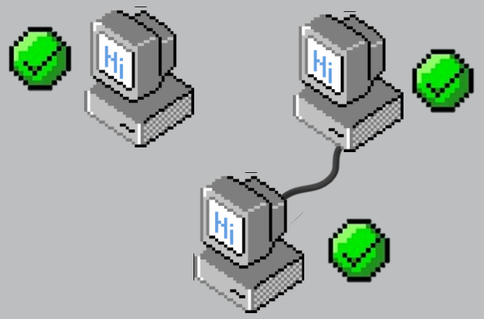
Los componentes del sistema pueden estar compuestos por diferentes tipos de hardware, sistemas operativos o redes, pero deben trabajar en conjunto sin problemas.

Distribución Geográfica



Los recursos pueden estar ubicados en lugares geográficos diferentes, lo que permite compartir cargas de trabajo y proveer servicios globales.

Autonomía



Cada nodo es autónomo y puede funcionar de manera independiente, incluso desconectado temporalmente del resto del sistema.

HISTORIA Y EVOLUCIÓN

DÉCADA DE 1960

DÉCADA DE 1970

DÉCADA DE 1980

HISTORIA Y EVOLUCIÓN

DÉCADA DE 1990

DÉCADA DE 2000

DÉCADA DE 2010 EN ADELANTE

VENTAJAS

1. Uso eficiente de recursos
2. Flexibilidad y modularidad
3. Costo eficiente
4. Paralelismo
5. Mejora en la continuidad del negocio

DESVENTAJAS

1. Complejidad
2. Dificultad en la sincronización
3. Latencia
4. Seguridad
5. Gestión de fallos compleja

CASOS DE USO

1. Servicios en la Nube (Cloud Computing)

2. Aplicaciones Web Globales (Alta concurrencia y disponibilidad)

3. Sistemas de Bases de Datos Distribuidas

4. Comercio electrónico y pagos digitales

5. Sistemas de Control Industrial y de IoT

6. Computación Científica y Big Data

CASOS DE APLICACIÓN (EJEMPLOS Y CASOS DE ÉXITO EN LA INDUSTRIA)

1. Facebook (Meta) – Sistema de almacenamiento distribuido TAO

2. Uber – Plataforma de movilidad en tiempo real

3. Spotify – Streaming musical basado en servicios distribuidos

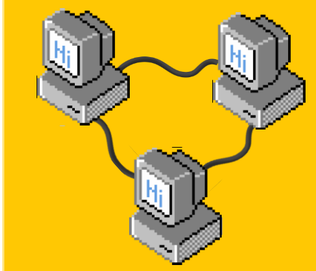
4.Google – Google File System (GFS) y MapReduce

5. Netflix – Microservicios y distribución global

6. Amazon – Comercio electrónico y AWS

TOPICOS ACANZADOS

1. Consenso Distribuido (Raft, Paxos):



Aplicaciones

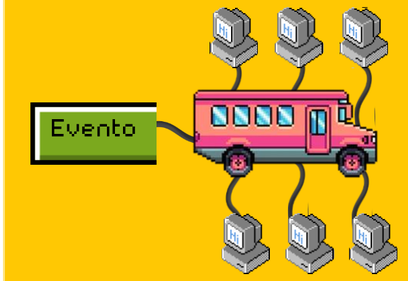
1. Coordinación de nodos.
2. Elección de líderes (master election).
3. Replicación consistente (ej: bases de datos distribuidas como CockroachDB).

TECNOLOGIAS

CONSUL

Zookeper

2. Sistemas Distribuidos Orientados a Eventos (EDA - Event-Driven Architecture):



Ventajas

- Alta desacoplamiento.
- Mejor escalabilidad y reactividad.
- Facilita el diseño de microservicios.

TECNOLOGIAS

kafka

KAFKA

RabbitMQ

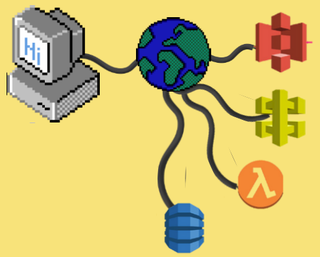
amazon SNS

Amazon SNS

Google Pub/Sub

TOPICOS ACANZADOS

3. Computación Sin Servidor (Serverless)



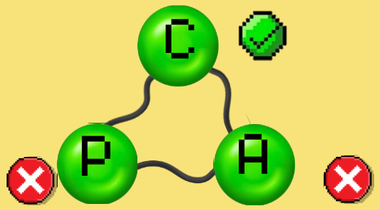
Modelo de ejecución donde el desarrollador no gestiona servidores. Solo escribe funciones que se ejecutan on-demand.

TECNOLOGIAS

Azure Functions

AWS Lambda

4. Teorema CAP (Consistencia, Disponibilidad, Tolerancia a Particiones)



En un sistema distribuido, donde no se pueden garantizar las tres propiedades al mismo tiempo:

- Consistencia
- Availability
- Partition Tolerance

TECNOLOGIAS

cassandra

DynamoDB

DynamoDB

RETOS EN SISTEMAS DISTRIBUIDOS

1. Consistencia de Datos:

- Problema: ¿Cómo asegurar que todos los nodos vean el mismo estado del sistema?
- Solución: Algoritmos de consenso (Raft, Paxos), sistemas con consistencia eventual.

2. Sincronización de Relojes

- Problema: Los eventos ocurren en diferentes momentos según cada reloj local.
- Solución: Protocolos como NTP, relojes lógicos (Lamport), relojes vectoriales.

3. Fallos de Red y Comunicación

- Problema: Los eventos ocurren en diferentes momentos según cada reloj local.
- Solución: Retransmisión, timeouts inteligentes, circuit breakers.

4. Heterogeneidad y Portabilidad

- Problema: Cómo hacer que los nodos trabajen juntos de manera eficiente
- Solución: Estandarización (REST/gRPC), contenedores (Docker).

5. Actualizaciones y Mantenimiento

- Problema: Evitar interrupciones al hacer mantenimiento.
- Solución: Despliegues blue/green, rolling updates, versionado de APIs.

6. Balanceo de Carga y Escalabilidad

- Problema: Algunos nodos reciben más carga que otros.
- Solución: Balanceadores de carga (NGINX, Envoy), autoscaling, replicación.

HERRAMIENTAS Y TECNOLOGIAS

comunicación y mensajeria

{REST}

Orquestación y contenedores

HELM

Bases de datos distribuidas

mongo DB

redis

Sistemas de Archivos Distribuidos

hadoop HDFS

Monitoreo y Observabilidad

ZIPKIN

Balanceadores de Carga y Proxies

NGINX

HAPROXY

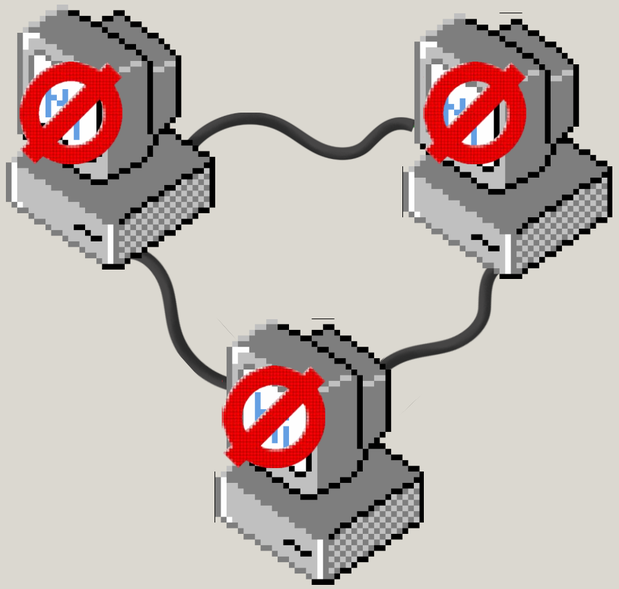
traefik

Kong

ZEROMQ
Run

ZERO MQ

Es una librería de mensajería asincrónica de alto rendimiento que permite la comunicación entre procesos (IPC), hilos, y sistemas distribuidos.

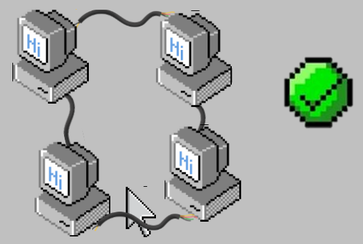


CARACTERISTICAS

Mensajería Asíncrona

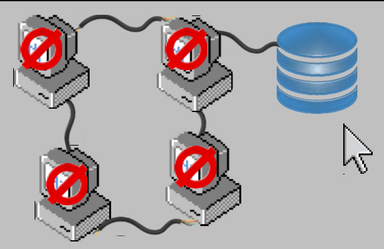
ZeroMQ permite a los componentes de una aplicación comunicarse de forma asíncrona

Sin intermediario



ZeroMQ no utiliza un broker central, como en otros sistemas de mensajería En su lugar, los componentes se conectan directamente entre sí.

Versatilidad



ZeroMQ se puede utilizar en una amplia variedad de contextos, desde sistemas de procesamiento de datos en tiempo real hasta aplicaciones de red,

Eficiencia

ZeroMQ está diseñado para ser rápido y eficiente, lo que lo hace adecuado para aplicaciones que requieren un rendimiento alto.

HISTORIA Y EVOLUCIÓN

HISTORIA Y EVOLUCIÓN

2013

2014-2018

2020

VENTAJAS

1. Alto rendimiento:
2. Sin necesidad de broker
3. Multiplataforma y multilenguaje
4. Patrones de comunicación integrados
5. Ligero y imbebible

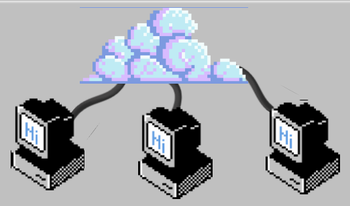
DESVENTAJAS

1. Sin persistencia integrada:
2. Manejo de errores manual
3. Curva de aprendizaje:
4. Requiere sincronización externa
5. No tiene monitoreo o administración web

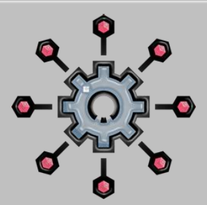
CASOS DE USO

1. Ahorro de tiempo en el desarrollo:

.2. Comunicación entre hilos o procesos dentro de una misma máquina



3. Balanceo de carga entre trabajadores



4. Desarrollo de microservicios

5. Sistemas distribuidos sin intermediarios

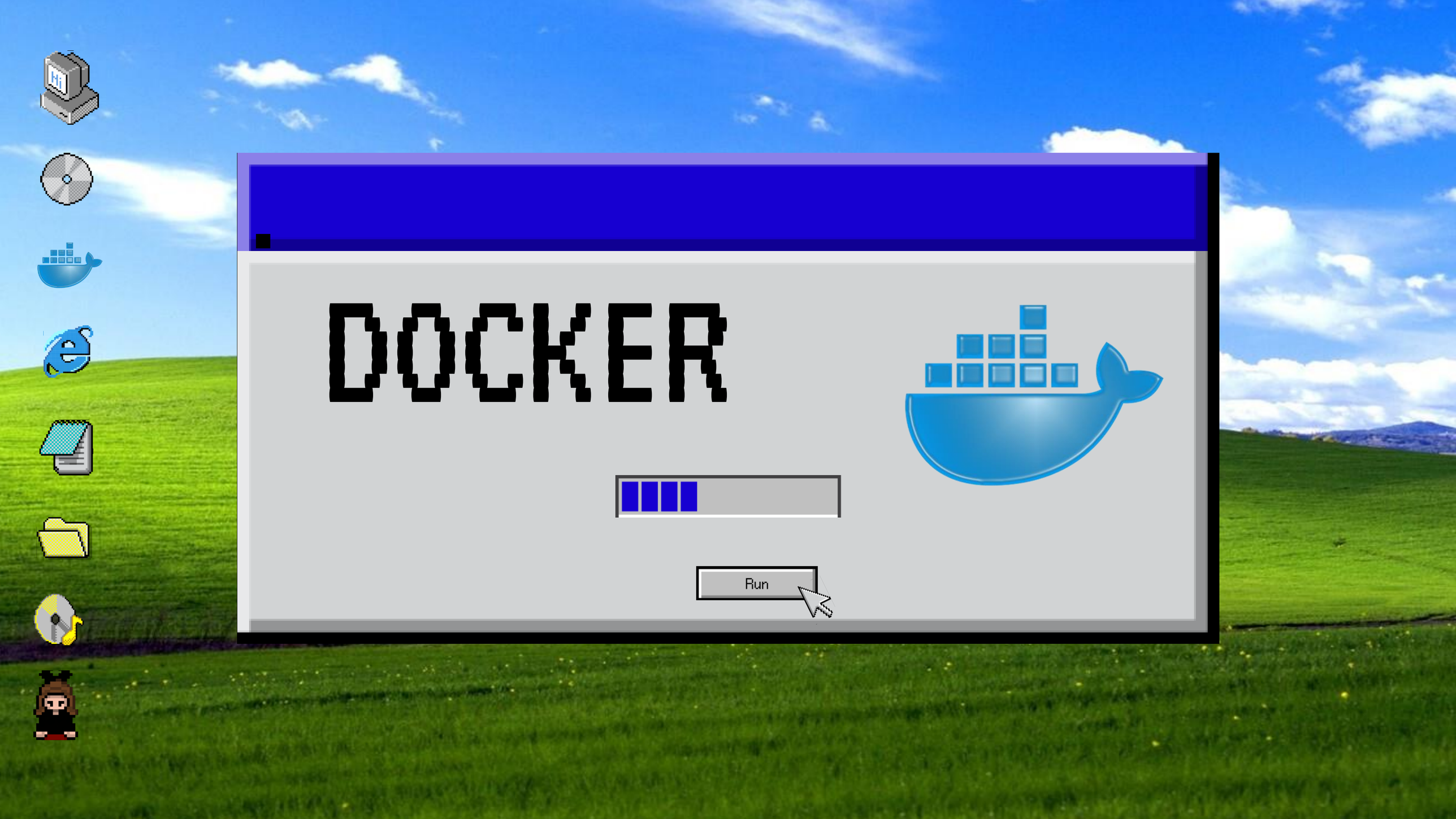
6. Sistemas de control distribuidos (automatización, robótica, IoT)

CASOS DE APLICACIÓN (EJEMPLOS Y CASOS DE ÉXITO EN LA INDUSTRIA)

1. CERN – Control de aceleradores de partículas

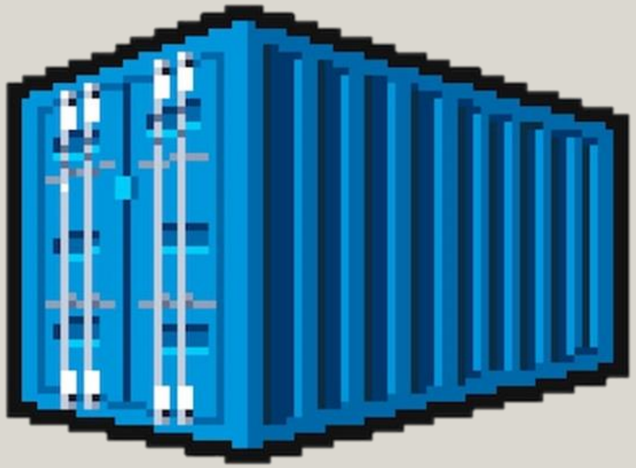
2. NASA – Prototipos de robótica y sistemas embebidos

3. Spotify – Streaming musical basado en servicios distribuidos


DOCKER
Run

DOCKER

Docker es una plataforma de software que permite crear, probar y ejecutar aplicaciones usando contenedores.Un contenedor es una especie de "caja" que empaca todo lo necesario para que una aplicación funcione

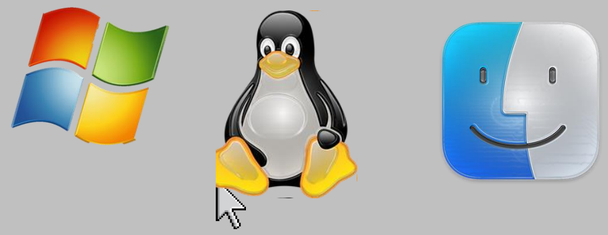


CARACTERISTICAS

Contenedores ligeros

Usan el mismo kernel del sistema operativo (a diferencia de las máquinas virtuales que necesitan un sistema completo).

Portabilidad



Un contenedor Docker se puede ejecutar en cualquier sistema que tenga Docker instalado, sin importar si es Linux, Windows o macOS

Aislamiento

Cada contenedor corre de forma independiente, esto significa que si un contenedor falla, no afecta a los demás.

Imágenes reutilizables

Docker usa imágenes como plantillas para crear contenedores. Cada usuario puede crear su propia imagen o usar imágenes públicas desde Docker Hub.

HISTORIA Y EVOLUCIÓN

2013

2015

2016-2025

VENTAJAS

1. Ahorro de tiempo en el desarrollo
2. Facilita el trabajo en equipo
3. Mejora la calidad del software
4. Despliegues más seguros y controlados:
5. Fomenta la automatización

DESVENTAJAS

1. Menor aislamiento que una VM
2. Curva de aprendizaje inicial
3. Gestión de datos persistentes
4. No apto para todas las aplicaciones
5. Problemas de seguridad

CASOS DE USO

1. Desarrollo en equipo

2. Pruebas automatizadas (CI/CD)

3. Despliegue en múltiples entornos

4. microservicios

5. Pruebas de versiones o tecnologías

6. Simulación de entornos complejos

CASOS DE APLICACIÓN (EJEMPLOS Y CASOS DE ÉXITO EN LA INDUSTRIA)

1. PayPal - Migró muchas aplicaciones a contenedores Docker.

2. ADP - : Usa Docker para entregar rápidamente aplicaciones financieras a sus clientes.

3. Spotify – Automatiza pruebas y despliegue de sus servicios musicales.

4.BBC - Utiliza Docker para la distribución de contenidos digitales y entornos de pruebas.

5. Netflix – Ejecuta pruebas A/B, microservicios y entornos de prueba con Docker.

6. ebay -Desarrolla y despliega microservicios usando Docker.

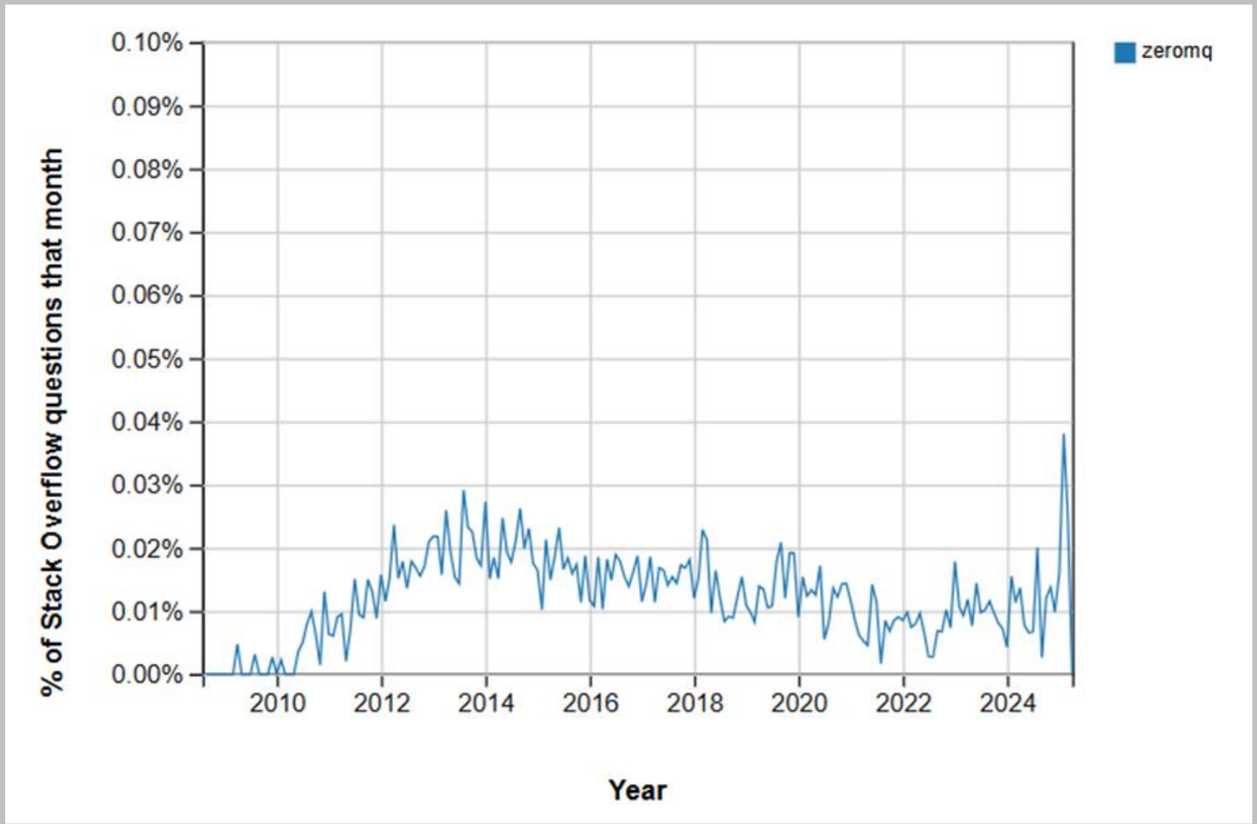
STACK

ØMQ

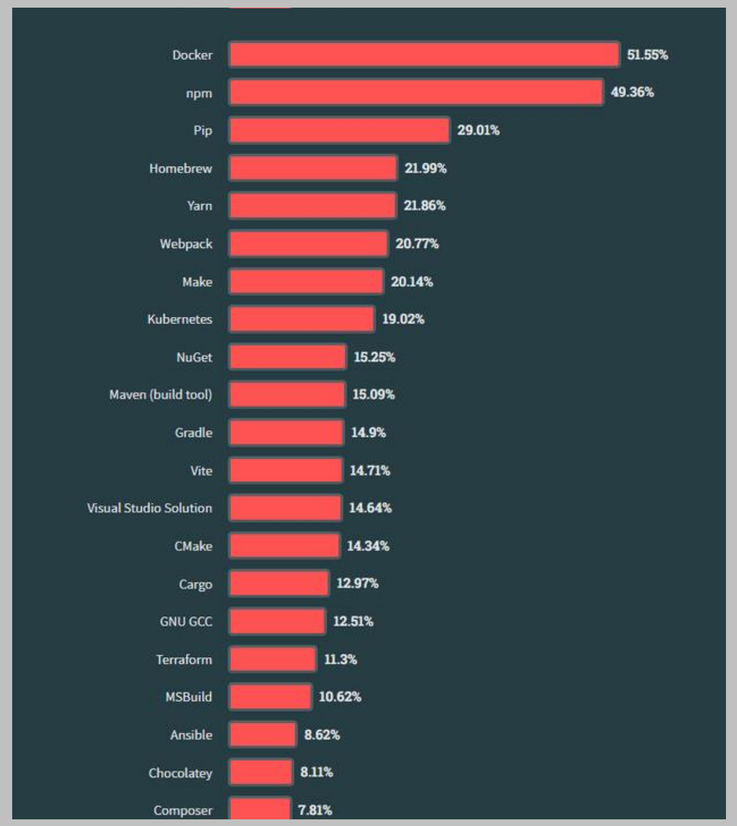
ZERO MQ

DOCKER

TENDENCIAS EN EL MERCADO ZEROMQ



TENDENCIAS EN EL MERCADO DOCKER



PRINCIPIOS SOLID VS ZEROMQ

Principios SOLID	Aplicación / Relevancia en zeroMQ
S – Single Responsibility	**Alto** – Es importante separar claramente las responsabilidades: enviar, recibir, serializar, etc. No se deben mezclar en una misma clase la lógica de negocio y la lógica de mensajería.
O – Open/Closed	**Moderado** – Es recomendable diseñar envoltorios (wrappers) o adaptadores sobre ZeroMQ que se puedan extender (ej. soporte para nuevos protocolos o patrones como pub/sub).
L – Liskov Substitution	**Limitado** – Dado que ZeroMQ opera a bajo nivel, este principio aplica sobre tus propias abstracciones (por ejemplo, un Mensaje debe poder sustituirse por una subclase sin alterar la semántica del canal de comunicación).
I – Interface Segregation	**Moderado** – Conviene definir interfaces específicas para productores, consumidores, brokers, etc., en lugar de una única interfaz de "cliente ZeroMQ". Esto facilita adaptabilidad y pruebas.
D – Dependency Inversion	**Alto** – Idealmente, tus componentes deben depender de interfaces (como IMessageBroker) en lugar de la implementación directa de ZeroMQ, lo que permite cambiar el backend de mensajería más fácilmente.

PRINCIPIOS SOLID VS DOCKER

Principios SOLID	Aplicación / Relevancia en zeroMQ
S – Single Responsibility	**Alto** – Docker fomenta contenedores con una única responsabilidad (ej. solo base de datos, solo backend), lo cual facilita el mantenimiento y escalabilidad.
O – Open/Closed	**Medio Alto** – Las imágenes pueden extenderse sin modificar las existentes (herencia de imágenes con Dockerfiles), permitiendo nuevas funciones sin alterar la base.
L – Liskov Substitution	**Medio** – Aunque es más teórico para programación orientada a objetos, en Docker se aplica al poder sustituir servicios por versiones compatibles sin romper el sistema.
I – Interface Segregation	**Medio Alto** – En el contexto de Docker, se refleja al dividir servicios en contenedores especializados, evitando dependencias innecesarias entre ellos.
D – Dependency Inversion	**Alto** –Docker permite invertir dependencias configurando entornos con inyecciones externas (ej. volúmenes, variables de entorno, redes), favoreciendo flexibilidad.

MATRIZ DE ANÁLISIS DE ATRIBUTOS DE CALIDAD VS ZEROMQ Y DOCKER

Atributo de Calidad	Evaluación en relación con ZeroMQ	Evaluación en relación con Docker
Disponibilidad	Depende de cómo se implemente. ZeroMQ no ofrece alta disponibilidad de forma nativa.	**Alta** – Docker permite replicar contenedores y usar herramientas externas (como Docker Swarm o Kubernetes) para alta disponibilidad y recuperación ante fallos.
Escalabilidad	**Alta** – Gracias a su arquitectura asíncrona y sin intermediarios (brokerless), escala fácilmente horizontalmente.	**Alta** – Facilita la creación y despliegue de múltiples instancias; escala horizontal con orquestadores como Kubernetes.
Desempeño	**Muy alta** – Uno de los puntos fuertes de ZeroMQ es su bajo overhead y velocidad (latencia muy baja).	**Media Alta** – Buen rendimiento, aunque depende del contenedor y recursos del host. El aislamiento puede introducir algo de sobrecarga.
Mantenibilidad	**Moderada** – Si no se encapsula bien, puede acoplar demasiado la lógica de negocio con el transporte. Aplicar buenas prácticas (como SOLID) es importante.	**Alta** – Los contenedores se basan en imágenes reproducibles y versionables, lo que facilita mantenimiento y actualizaciones.
Portabilidad	**Alta** – Compatible con múltiples plataformas y lenguajes (C, C++, Java, Python, etc.).	**Muy alta** – Corre en cualquier sistema con Docker instalado (Windows, Linux, Mac); entornos consistentes garantizados por las imágenes.
Seguridad	**Limitada** – ZeroMQ no incluye cifrado ni autenticación por defecto. Se recomienda usar ZAP o CurveZMQ si se requiere seguridad.	**Media** – Ofrece aislamiento, control de red y permisos, pero requiere configuraciones adicionales para seguridad avanzada
Interoperabilidad	**Buena** – Puede comunicarse con aplicaciones en distintos lenguajes y puede integrarse con otros sistemas vía sockets y patrones de mensajería.	**Alta** – Puede contener y comunicar cualquier tipo de aplicación a través de redes, APIs, volúmenes compartidos, etc.
Confiabilidad	**Moderada** – No garantiza entrega de mensajes por sí solo (no tiene persistencia incorporada). Se debe diseñar lógica adicional si se requiere fiabilidad.	**Alta** – Con buenas prácticas (como supervisión, volúmenes persistentes y orquestadores), puede asegurar despliegues consistentes y recuperación ante fallos.

MATRIZ DE ANÁLISIS DE TÁCTICAS VS ZEROMQ

Atributo de Calidad	Táctica arquitectónica	Aplicación en ZeroMQ
Disponibilidad	Failover activo-pasivo	No soportado nativamente. Se debe diseñar con lógica externa para reconexión o balanceo manual.
	Heartbeat / detección de fallos	Se puede implementar usando sockets REQ/REP con timeouts o patrones PUB/SUB.
Desempeño	Colas asincrónicas	Patrón nativo de ZeroMQ (PUSH/PULL), ideal para desacoplar productores y consumidores.
	Reducir overhead de comunicación	ZeroMQ está diseñado para minimizar overhead: sin intermediarios ni cabeceras pesadas.
Escalabilidad	Replicación y distribución de carga	Patrón PUSH/PULL o DEALER/ROUTER permite distribuir carga entre múltiples nodos.
	Componentes sin estado	ZeroMQ se adapta bien a arquitecturas sin estado (stateless), facilitando escalado horizontal
Mantenibilidad	Separación de responsabilidades	Requiere aplicar SRP/SOLID externamente; ZeroMQ no impone modularidad.
	Encapsulamiento	Puede encapsularse en interfaces abstractas (IMessenger, IMessageBus, etc.).
Seguridad	Autenticación y cifrado de mensajes	No es por defecto. Se requiere implementar CurveZMQ o usar un proxy con TLS.
Confiabilidad	Persistencia de mensajes	ZeroMQ no guarda mensajes (no es un message broker). Debe complementarse con otras capas.

MATRIZ DE ANÁLISIS DE TÁCTICAS VS DOCKER

Atributo de Calidad	Táctica arquitectónica	Aplicación en Docker
Disponibilidad	Failover activo-pasivo	Requiere orquestador (como Kubernetes) para lograr alta disponibilidad.
	Heartbeat / detección de fallos	Soportado con HEALTHCHECK y manejado automáticamente por orquestadores.
Desempeño	Colas asincrónicas	No nativo. Requiere herramientas externas como Redis, RabbitMQ o Kafka.
	Reducir overhead de comunicación	Contenedores ligeros reducen el overhead respecto a máquinas virtuales.
Escalabilidad	Replicación y distribución de carga	Compatible con Swarm/Kubernetes para balanceo y escalado automático.
	Componentes sin estado	Fomenta diseño stateless; ideal para arquitecturas escalables.
Mantenibilidad	Separación de responsabilidades	Docker permite aislar servicios/lógicas en contenedores distintos.
	Encapsulamiento	Cada contenedor incluye su propio entorno y dependencias.
Seguridad	Autenticación y cifrado de mensajes	No nativo. Se debe configurar TLS manualmente y aplicar buenas prácticas.
Confiabilidad	Persistencia de mensajes	Requiere volúmenes, bases de datos o message brokers externos para persistencia.

Categoría	Patrón Arquitectonico	Aplicación con ZeroMQ
Distribuidos	Broker	ZeroMQ es brokerless por defecto, pero se puede implementar un broker personalizado si es necesario.
	Peer-to-peer (P2P)	**Muy compatible** – ZeroMQ permite comunicación directa entre nodos.
	Event-Driven Architecture (EDA)	**Compatible** – Con patrones PUB/SUB, ZeroMQ facilita sistemas basados en eventos.
	Microservicios	**Compatible** – Facilita comunicación ligera entre servicios, ideal para este estilo
Mensajería	Message Queue	Con PUSH/PULL, ZeroMQ implementa colas de trabajo (pero sin persistencia).
	Publicador/Subscriptor	**Nativo** – Patrón incluido en la librería.
	Request/Reply	**Soportado** – Patrón REQ/REP bien definido.
	Bus de mensajes	Posible pero requiere esfuerzo. Se puede construir un bus con múltiples sockets.
Diseño de software	Adapter	**Muy útil** – ZeroMQ puede ser adaptado a otras interfaces con facilidad.
	Facade	**Recomendado** – Útil para ocultar la complejidad de la mensajería bajo una interfaz clara.
	Observer	Compatible – Patrón Observer se puede mapear al PUB/SUB de ZeroMQ.
	Proxy	Se puede implementar para ocultar detalles de comunicación entre procesos.

MATRIZ DE ANÁLISIS DE PATRONES VS ZEROMQ

ØMQ

Patrón Arquitectonico	Aplicación con Docker
Microservicios	**Altamente compatible** - Docker permite aislar cada microservicio en su contenedor, facilitando despliegue y escalado.
Monolito desacoplado	**Compatible** - Un monolito puede ser contenedorizado fácilmente, aunque no se aprovechan todos los beneficios de Docker.
Event-Driven Architecture (EDA)	**Compatible con herramientas externas** - Docker no maneja eventos por sí solo, pero puede alojar brokers de eventos como Kafka o RabbitMQ.
Pipeline / ETL	**Compatible** - Útil para aislar etapas del procesamiento de datos en contenedores independientes
CQRS / Event Sourcing	**Requiere herramientas externas** - Docker puede contener componentes CQRS, pero necesita bases de datos/event stores externos.
Batch processing	**Compatible** -Ideal para tareas por lotes encapsuladas en contenedores que se ejecutan y destruyen automáticamente.
Serverless	**Parcialmente compatible** - Docker puede ejecutar funciones empaquetadas, pero plataformas como AWS Lambda abstraen más el entorno.
12-Factor App	**Altamente compatible** - Docker sigue los principios 12-Factor: portabilidad, configuración por entorno, procesos aislados, etc.
Service Mesh	**Compatible con orquestadores** - Requiere herramientas adicionales (como Istio o Linkerd) que corren dentro de entornos Docker/Kubernetes.

MATRIZ DE ANÁLISIS DE PATRONES VS DOCKER

DIAGRAMAS

Run

DIAGRAMA DE CONTEXTO

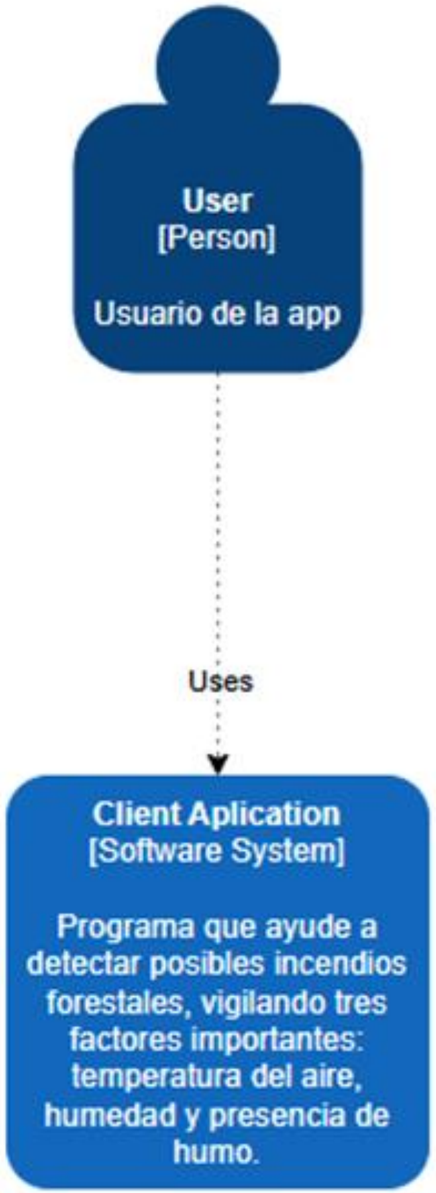


DIAGRAMA DE ALTO NIVEL

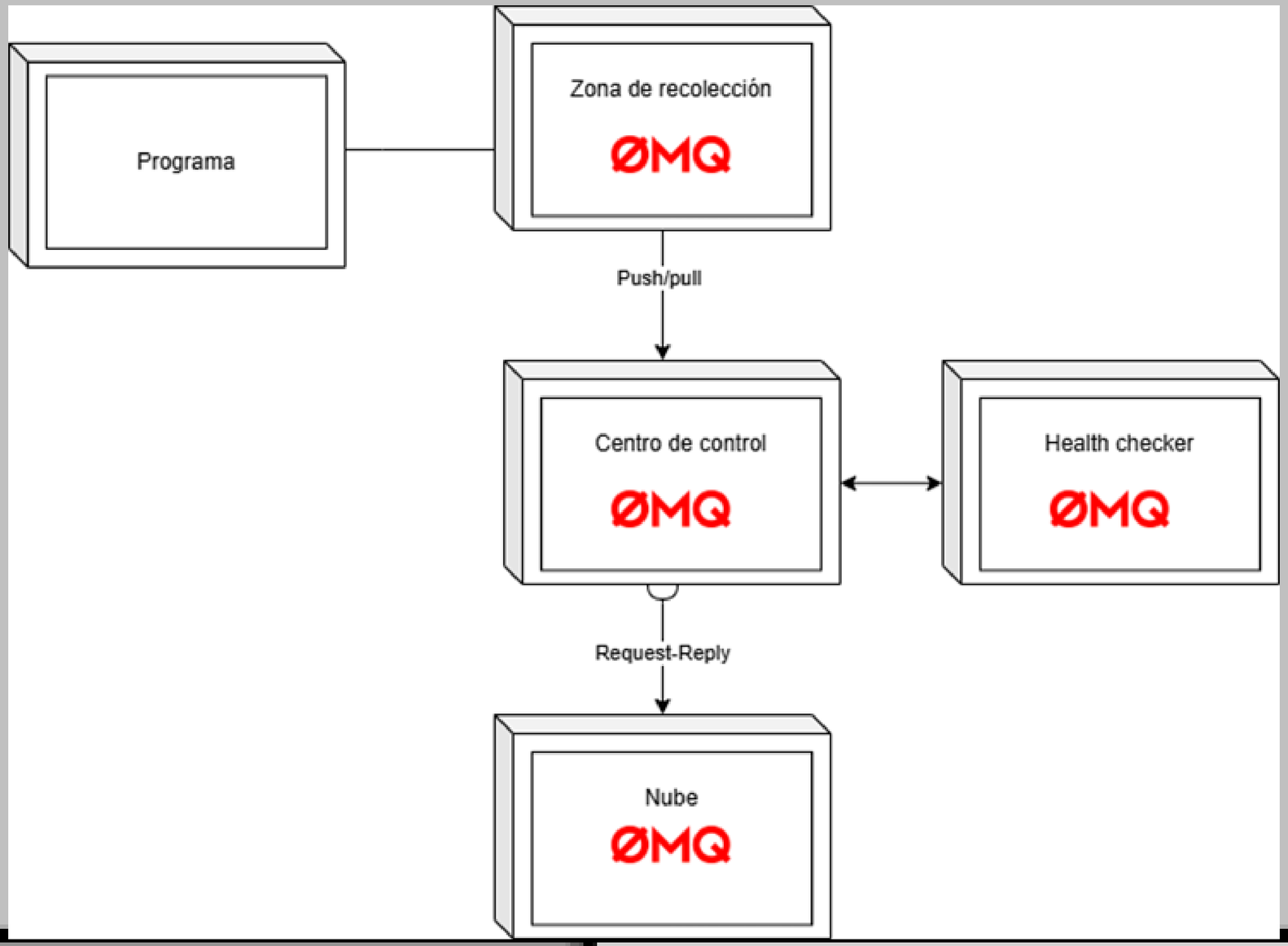


DIAGRAMA DINAMICO

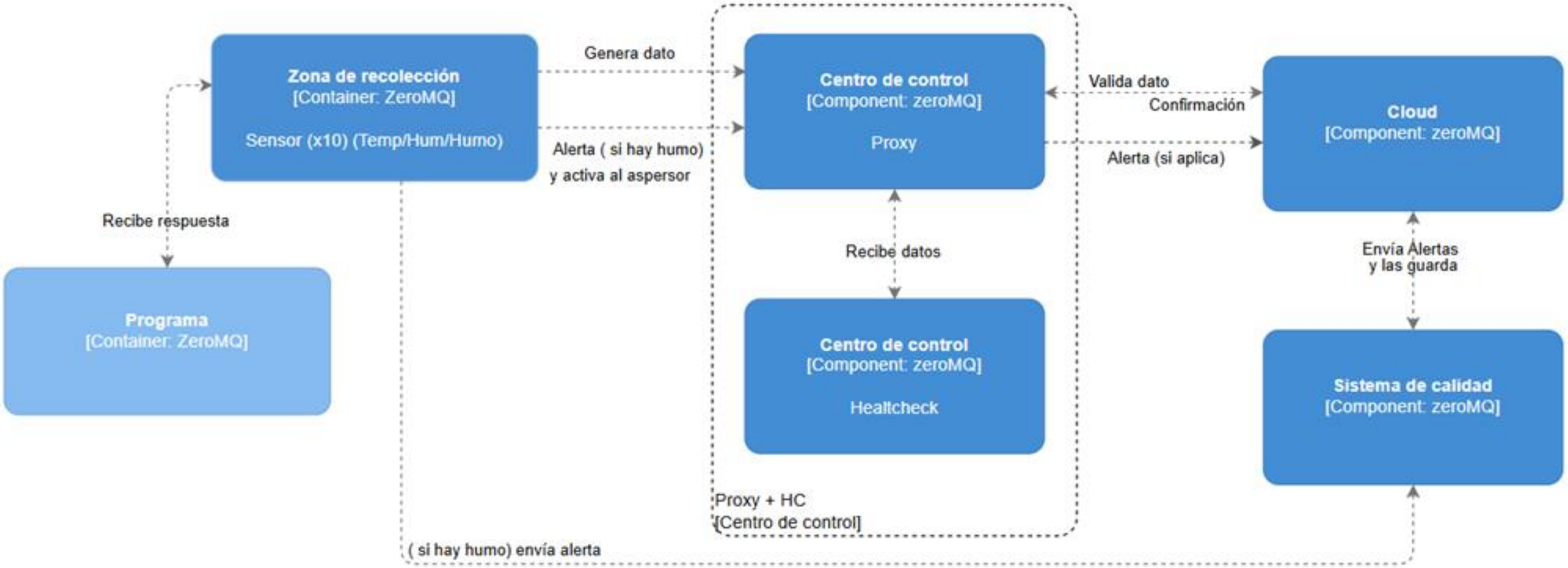


DIAGRAMA DE DESPLIEGUE

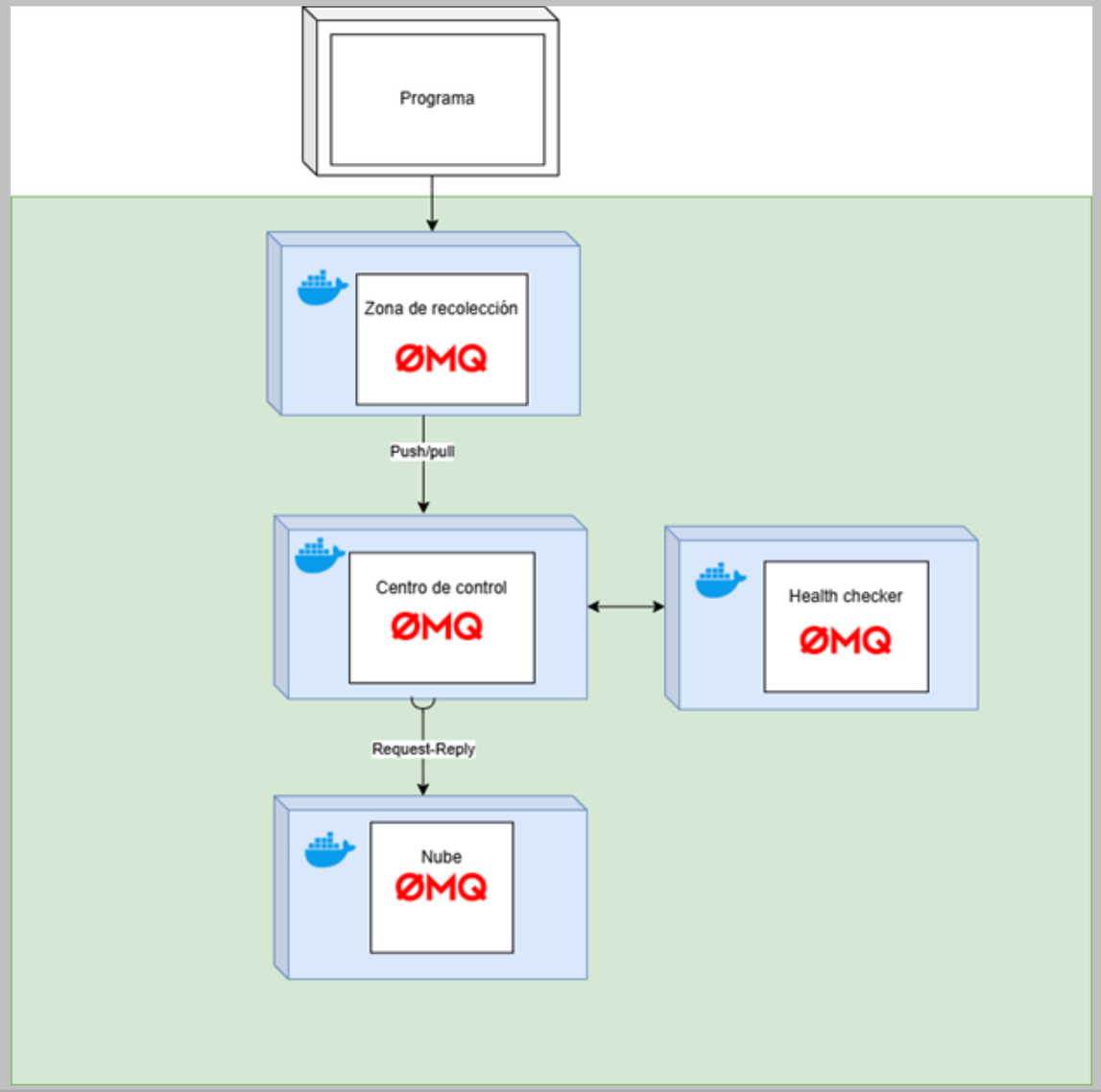


DIAGRAMA DE PAQUETES UML

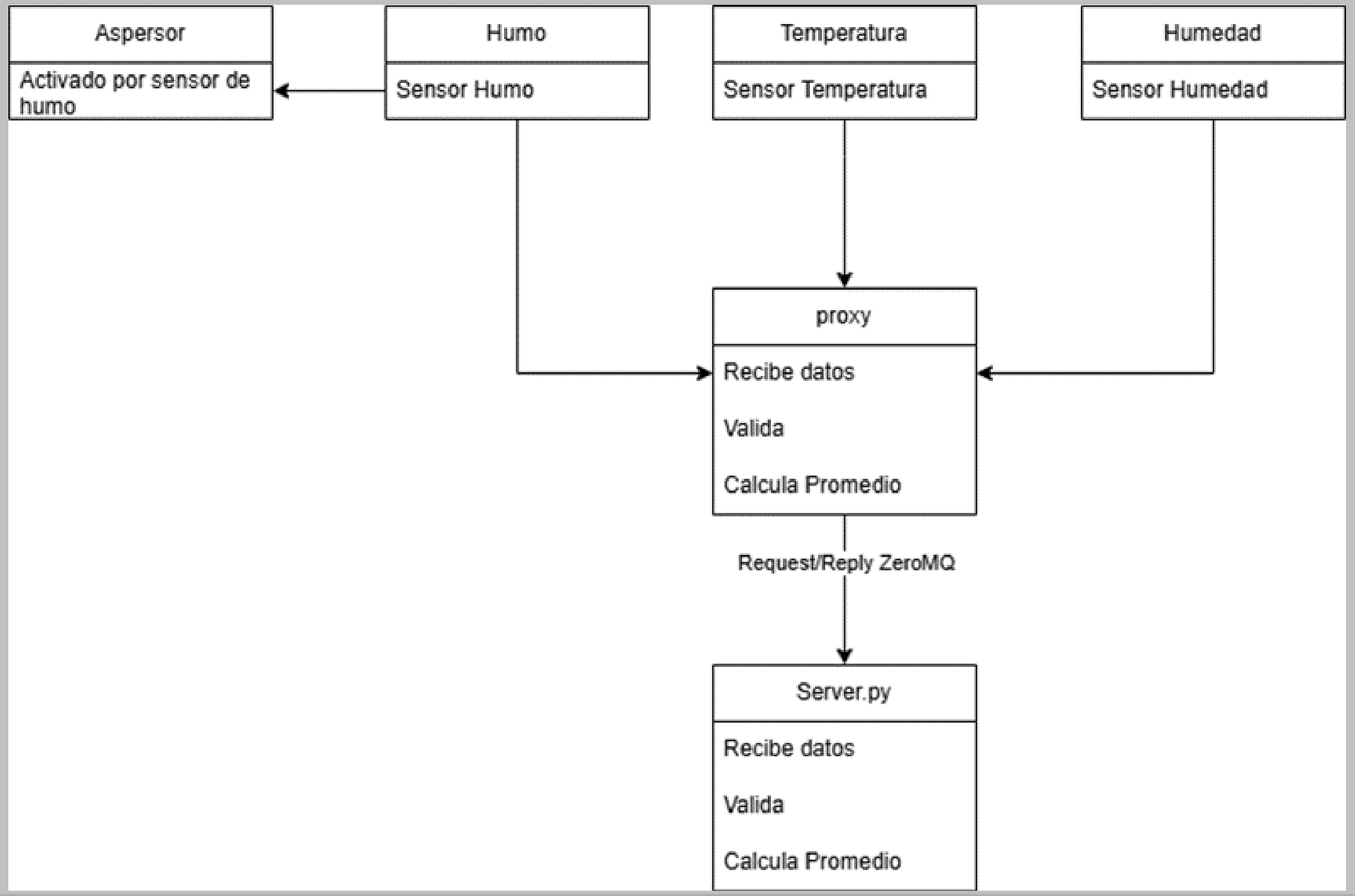


DIAGRAMA DE COMPONENTES C4

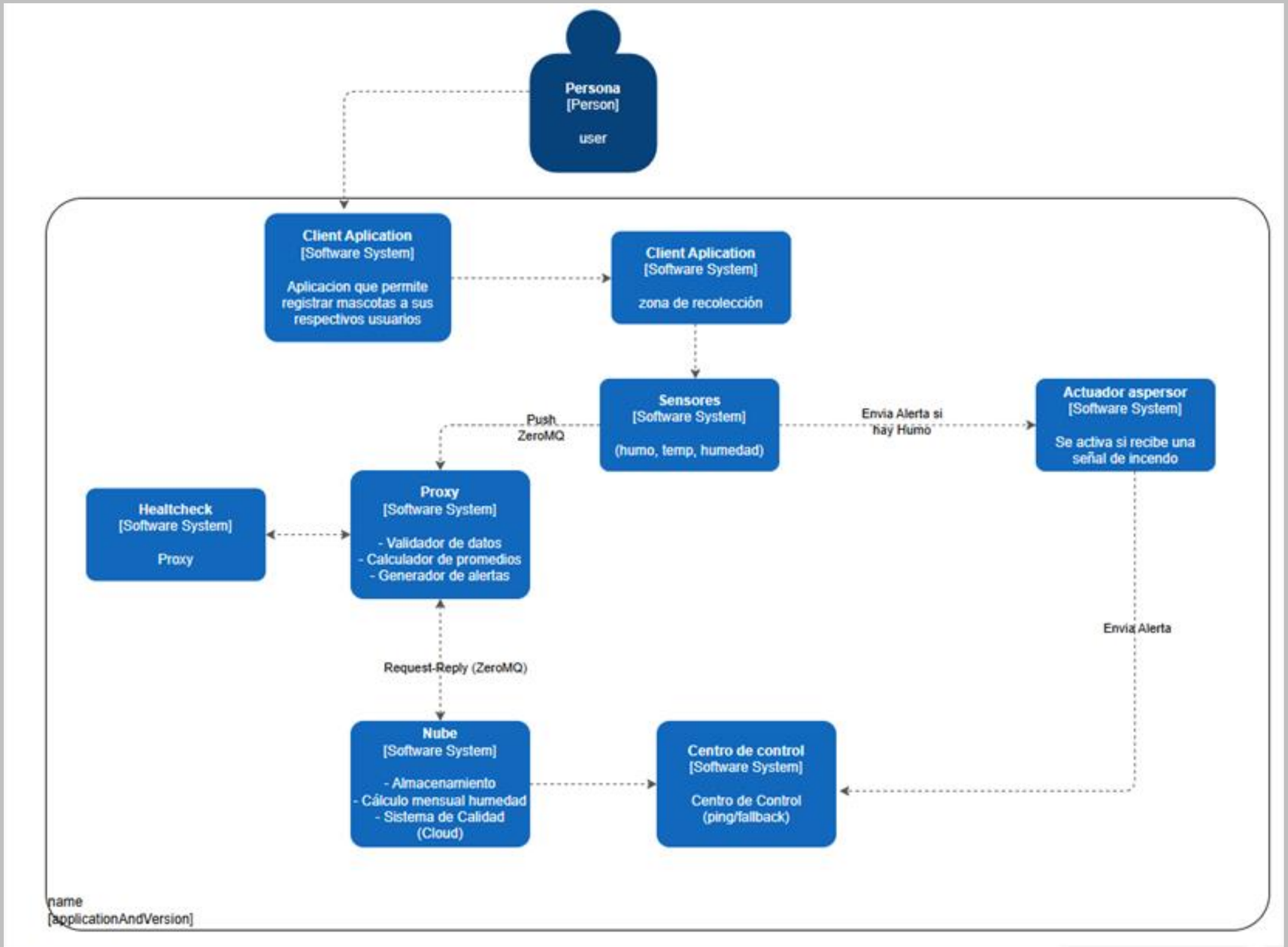


DIAGRAMA DE CONTENEDORES

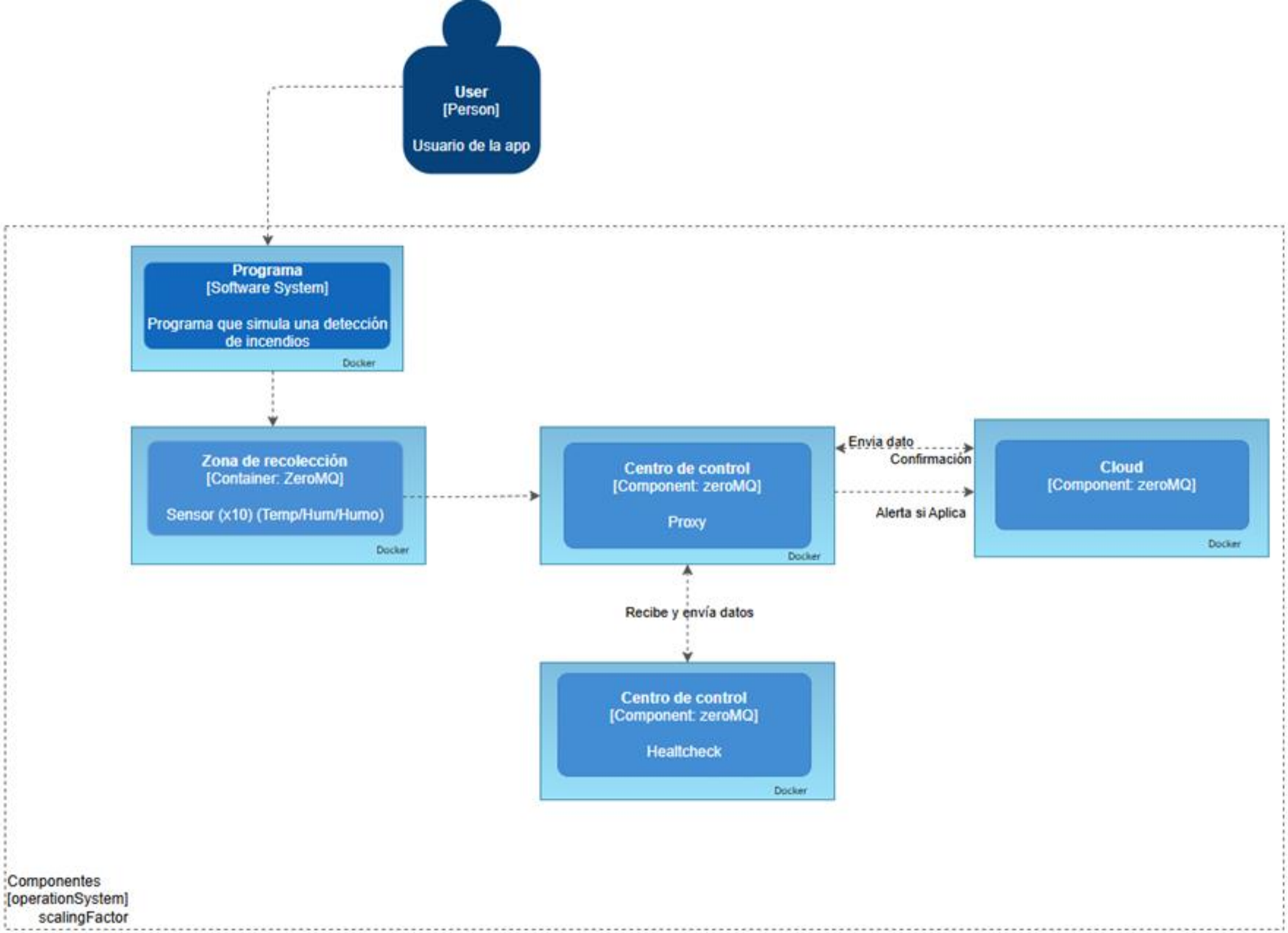
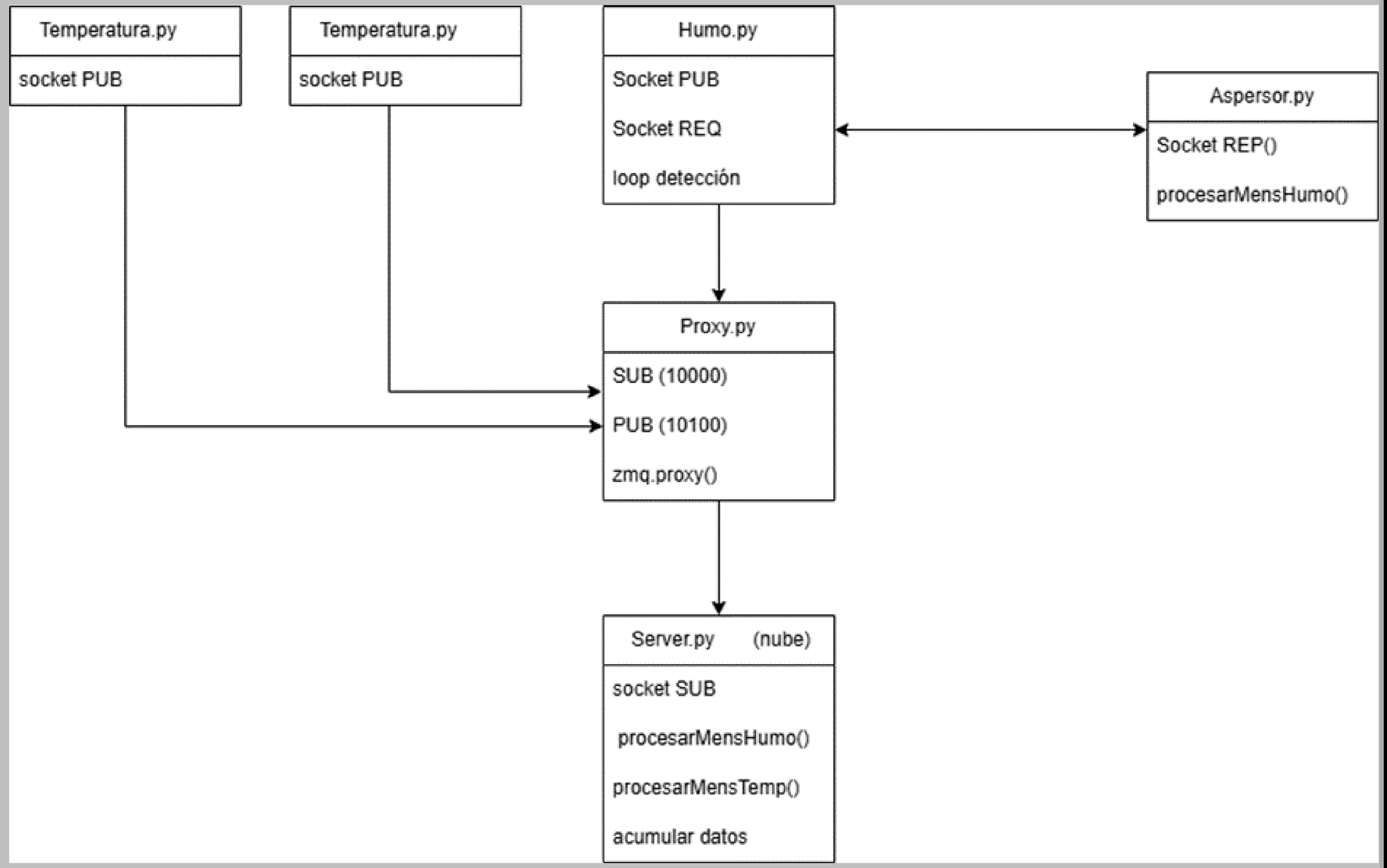


DIAGRAMA DE CODIGO



File Edit Selection View Go Run Terminal Help

Detector-de-Incendios

EXPLORER

DETECTOR-DE-INCENDIOS

- Aspersor
  - Aspersor.py
  - Dockerfile
  - requirements.txt
- Humedad
  - Dockerfile
  - Humedad.py
  - requirements.txt
- Humo
  - Dockerfile
  - Humo.py
  - requirements.txt
- proxy
  - Dockerfile
  - proxy.py
  - requirements.txt
- server
  - Dockerfile
  - requirements.txt
  - server.py
- Temperatura
  - Dockerfile
  - requirements.txt
  - Temperatura.py
- docker-compose.yml

Humedad.py 1 | Humo.py 1 | server.py 1 | proxy.py 1 | Temperatura.py 1 | Aspersor.py 1

server > server.py > procesarMensTemp

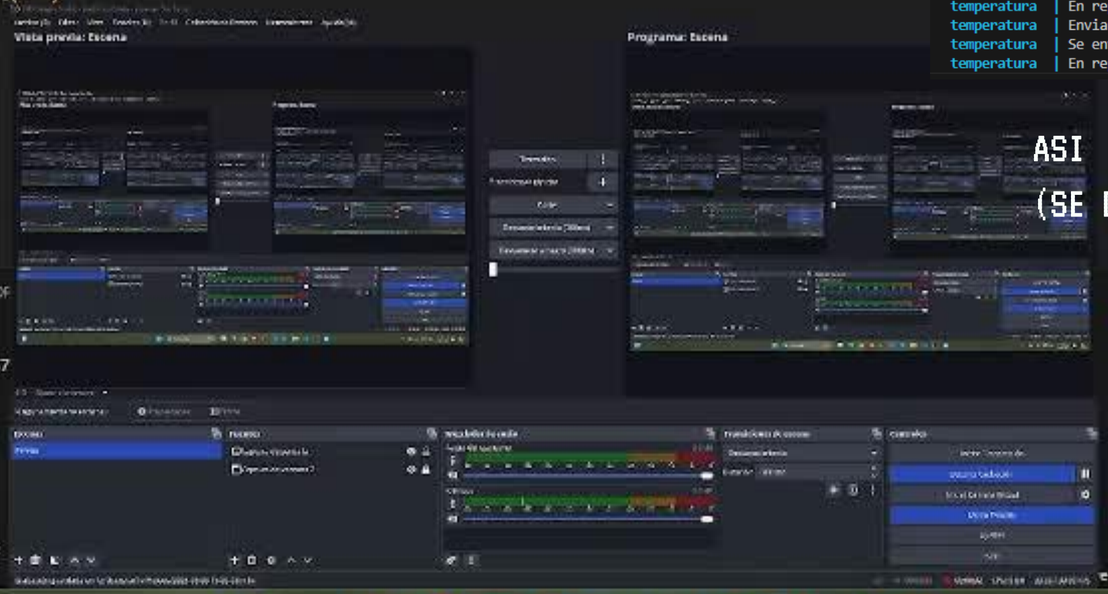
```python
def procesarMensHume(msg):

    hume = float(msg)
    if hume > 0.7 and hume < 1:
      print("Reporte de humedad", msg)
      acumuladorHume = acumuladorHume + hume
      contadorHume = contadorHume +1
      if contadorHume >= 10:
        promedio = acumuladorHume / contadorHume
        print ("La humedad promedio es:", promedio, time.strftime("%c"))
        acumuladorHume = 0.0
        contadorHume = 0
    else:
        print("Mensaje no valido")

def procesarMensAsper(msg):
    """
    docstring
    """
    print("Reporte de Aspersor", msg)


# Process 5 updates
while True:
  print("Escuchando ...")
  total_value = 0
  topic, msg = socket.recv_string().split(" ")
  print("Recibido un mensaje")
  if topic == "Humo":
    procesarMensHumo(msg)
  elif topic == "Temperatura":
    procesarMensTemp(msg)
  elif topic == "Humedad":
    procesarMensHume(msg)
  elif topic == "Aspersor":
    procesarMensAsper(msg)
```

PROBLEMS 6 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

```
humedad     | En reposo
humedad     | Enviando reporte de temperatura ...
humedad     | Se enviò el mensaje al proxy  Humedad:  0.7797486313758037
humedad     | En reposo
humedad     | Enviando reporte de temperatura ...
humedad     | Se enviò el mensaje al proxy  Humedad:  0.7867500324138169
humedad     | En reposo
humedad     | Enviando reporte de temperatura ...
humedad     | Se enviò el mensaje al proxy  Humedad:  0.8694742338022496
humedad     | En reposo
humedad     | Enviando reporte de temperatura ...
humedad     | Se enviò el mensaje al proxy  Humedad:  0.985047562222902
humedad     | En reposo
humedad     | Enviando reporte de temperatura ...
humedad     | Se enviò el mensaje al proxy  Humedad:  0.9655814291245803
humedad     | En reposo
humedad     | Enviando reporte de temperatura ...
humedad     | Se enviò el mensaje al proxy  Humedad:  0.9255215691870192
humedad     | En reposo
```

PROBLEMS 6 | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS

```
temperatura | En reposo
temperatura | Enviando reporte de temperatura ...
temperatura | Se enviò el mensaje al proxy  Temperatura:  20.315398403095582
temperatura | En reposo
temperatura | Enviando reporte de temperatura ...
temperatura | Se enviò el mensaje al proxy  Temperatura:  26.911518254107158
temperatura | En reposo
temperatura | Enviando reporte de temperatura ...
temperatura | Se enviò el mensaje al proxy  Temperatura:  21.46229183224878
temperatura | En reposo
temperatura | Enviando reporte de temperatura ...
temperatura | Se enviò el mensaje al proxy  Temperatura:  24.61702327945727
temperatura | En reposo
temperatura | Enviando reporte de temperatura ...
temperatura | Se enviò el mensaje al proxy  Temperatura:  23.723696934386147
temperatura | En reposo
temperatura | Enviando reporte de temperatura ...
temperatura | Se enviò el mensaje al proxy  Temperatura:  14.327754060459888
temperatura | En reposo
```



ASI SE VEN LAS DEMAS ALERTAS
(SE DEMORA UN POCO EN APARECER)

PROBLEMS 6 | OUTPUT | DEBUG CONSOLE | TERMINAL | POR

```
server      | Recibido un mensaje
server      | Reporte de humedad 0.9261638172086957
server      | Escuchando ...
server      | Recibido un mensaje
server      | Reporte de humo:  False
server      | Escuchando ...
server      | Recibido un mensaje
```

View in Docker Desktop | View Config | Enable

docker-compose

OUTLINE

TIMELINE

main | 0 6 | Git Graph | Ln 41, Col 10 | Spaces: 2 | UTF-8 | LF | Python | 3.14.0

Búsqueda

3:08 p. m.
3/06/2025

CONCLUSIONES Y LECCIONES APRENDIDAS

Run

G R A C I A S