SISTEMAS DISTRIBUIDOS
ZERO MQ

SOFIA CESPEDES VARGAS

ARQUITECTURA DE SOFTWARE

FACULTAD DE INGENIERÍA

PONTIFICIA UNIVERSIDAD JAVERIANA

BOGOTA D.C

27/05/2025

Introducción

Sistemas Distribuidos

Un sistema distribuido 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. El medio de comunicación de estas computadoras para coordinar sus acciones es a través de una red en la cual logran compartir recursos (hardware, software o información). Los sistemas distribuidos se diseñan para ofrecer escalabilidad, tolerancia a fallos, disponibilidad y eficiencia y aunque son independientes , ante el usuario son un sistema único y coherente.

Características

Un sistema distribuido posee una serie de características distintivas que lo diferencian de los sistemas centralizados. Estas características permiten que los sistemas distribuidos sean escalables, robustos y adecuados para entornos modernos como la computación en la nube, microservicios y aplicaciones web distribuidas.

1. **Transparencia**

   El sistema debe ocultar a los usuarios y a los programadores el hecho de que sus procesos y recursos están distribuidos. Existen diferentes tipos de transparencia:

- **Transparencia de acceso:** los usuarios acceden a recursos sin saber su ubicación.
- **Transparencia de ubicación**: el recurso puede estar en cualquier máquina.
- **Transparencia de concurrencia:** múltiples usuarios pueden usar el mismo recurso sin interferencias.
- **Transparencia de replicación:** el usuario no nota si el recurso está replicado.
- **Transparencia de fallos:** el sistema continúa funcionando a pesar de fallos parciales.

2. **Escalabilidad**

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

3. **Tolerancia a fallos**

   El sistema debe poder recuperarse de fallos parciales, como la caída de un nodo o pérdida temporal de conectividad, sin comprometer la disponibilidad global.

4. **Concurrencia**

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

5. **Heterogeneidad**

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

6. **Distribución geográfica**

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

7. **Autonomía**

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

Historia y evolución

Los sistemas distribuidos están ligados al desarrollo de las redes de computadoras, los sistemas operativos y la necesidad de compartir recursos computacionales de forma eficiente.

Evolución:

- **Década de 1960 – Primeros conceptos**

  Durante esta época se desarrollaron los primeros sistemas de tiempo compartido, donde múltiples usuarios podían interactuar con un solo computador central desde terminales remotas. Aunque no eran sistemas distribuidos en el sentido moderno, introdujeron el concepto de computación compartida.

  Ejemplo: Proyecto MULTICS (Multiplexed Information and Computing Service), pionero en el uso compartido de recursos.

- **Década de 1970 – Nacimiento de redes de computadoras**

  Con el surgimiento de redes como ARPANET, los investigadores comenzaron a experimentar con la interconexión de múltiples sistemas, estructurando las bases de los sistemas distribuidos. Se introducen conceptos como el procesamiento distribuido, donde varias máquinas colaboran para resolver tareas y aparecen los primeros protocolos de red, como TCP/IP.

- **Década de 1980 – Consolidación y teoría formal**

  Se popularizan las estaciones de trabajo y las redes locales (LAN), lo que permite implementar sistemas distribuidos en entornos académicos e industriales. Se desarrollan sistemas como NFS (Network File System) de Sun Microsystems, permitiendo compartir archivos entre sistemas además surgen los primeros sistemas de archivos distribuidos y RPC (Remote Procedure Call).

- **Década de 1990 – Middleware y computación cliente-servidor**

  La arquitectura cliente-servidor se vuelve dominante, facilitando la comunicación entre aplicaciones distribuidas.También aparece el concepto de middleware, software que facilita la comunicación entre componentes distribuidos heterogéneos. En ese momento las tecnologías más destacadas eran: CORBA, DCOM, Java RMI y comienza la integración de sistemas heterogéneos.

- **Década de 2000 – Web, cloud y servicios**

  La expansión de Internet impulsa los sistemas distribuidos a gran escala. Surgen arquitecturas como SOA (Service-Oriented Architecture) y tecnologías como web services y REST, permitiendo interoperabilidad global. Surge la aparición de sistemas de almacenamiento distribuido (Ej.: Google File System) y la introducción de sistemas de cómputo en la nube, como Amazon Web Services (AWS).

- **Década de 2010 en adelante – Microservicios y computación en la nube**

  Los sistemas distribuidos evolucionan hacia arquitecturas de **microservicios**, donde las aplicaciones se dividen en componentes pequeños e independientes que se comunican mediante APIs.

  - Tecnologías clave: Docker, Kubernetes, gRPC, Apache Kafka, NoSQL distribuido.
  - Popularización de la computación sin servidor (serverless) y el uso de contenedores.
  - Aumento del uso de sistemas distribuidos en tiempo real y edge computing.

Ventajas y desventajas de los Sistemas Distribuidos

Ventajas	**Desventajas**
Uso eficiente de recursos: Permite aprovechar recursos distribuidos en diferentes ubicaciones.	Complejidad: El diseño, desarrollo y mantenimiento es más complejo.
Flexibilidad y modularidad: Fácil incorporación de nuevos servicios o tecnologías.	Dificultad en la sincronización: Mantener coherencia entre nodos es difícil.
Costo eficiente: Se puede usar hardware económico distribuido.	Latencia: La comunicación entre nodos puede introducir retardos.
Paralelismo: Varias tareas pueden ejecutarse en paralelo en diferentes nodos, acelerando el procesamiento.	Seguridad: Superficie de ataque mayor debido a la exposición de múltiples nodos.
Mejora en la continuidad del negocio: Gracias a la replicación y redundancia, se minimiza el impacto de fallos críticos y se garantiza la continuidad operativa.	Gestión de fallos compleja: Detectar y manejar errores requiere mecanismos especializados.

Casos de Uso (Situaciones y/o problemas donde se pueden aplicar)

Los sistemas distribuidos son implementados en muchos contextos actuales debido a su capacidad de escalar, tolerar fallos y ofrecer servicios de forma eficiente. Algunas situaciones concretas y problemas típicos donde su implementación es muy efectiva son los siguientes:

1. **Servicios en la Nube (Cloud Computing)**

   Hay algunas empresas que necesitan infraestructura flexible para alojar aplicaciones, bases de datos, y servicios web una solución es evitar el uso de servidores físicos dedicados que son costosos y difíciles de escalar, es mejor utilizar aplicaciones que les permita manejar todo en la nube.

   Ejemplo: Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform usan sistemas distribuidos para ofrecer servicios bajo demanda.

2. **Aplicaciones Web Globales (Alta concurrencia y disponibilidad)**

   Plataformas como redes sociales, e-commerce o streaming con millones de usuarios concurrentes necesitan de alta disponibilidad, balanceo de carga y replicación de datos en múltiples regiones.

   Ejemplo: Facebook, Netflix y Amazon utilizan arquitecturas distribuidas basadas en microservicios para escalar horizontalmente.

3. **Sistemas de Bases de Datos Distribuidas**

   Empresas que manejan grandes volúmenes de datos desde múltiples sedes o regiones.
   necesitan de acceso concurrente y rápido a los datos desde distintas ubicaciones sin pérdida de consistencia.

   Ejemplo: MongoDB, Cassandra, Google Spanner, que replican y distribuyen datos en múltiples nodos.

4. **Comercio electrónico y pagos digitales**

   Procesamiento de miles o millones de transacciones financieras por segundo. requieren tolerancia a fallos, disponibilidad continua y protección ante caídas de servidores.

   Ejemplo: Plataformas como PayPal, Stripe o MercadoPago usan arquitecturas distribuidas con redundancia.

5. **Sistemas de Control Industrial y de IoT (Internet de las Cosas)**

   Sensores y dispositivos distribuidos geográficamente que recolectan datos en tiempo real necesitan procesar y reaccionar a eventos desde múltiples ubicaciones simultáneamente.

Ejemplo: Sistemas de monitoreo ambiental, ciudades inteligentes, fábricas conectadas (Industry 4.0).

6. **Computación Científica y Big Data**

   Procesamiento de grandes volúmenes de datos para investigación o aprendizaje automático requieren alto poder de procesamiento y almacenamiento distribuido.

   Ejemplo: Uso de clústeres con Hadoop o Spark para análisis de datos genómicos, simulaciones científicas, o meteorología.

Casos de aplicación (Ejemplos y casos de éxito en la industria)

Los sistemas distribuidos son la base tecnológica de muchas empresas líderes en sus respectivos sectores. A continuación, se presentan ejemplos reales de cómo estos sistemas son aplicados en la industria para resolver desafíos técnicos y escalar globalmente.

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

- Problema: Google necesitaba procesar y almacenar enormes cantidades de datos provenientes de su motor de búsqueda y otros servicios.
- Solución distribuida: Desarrollaron GFS, un sistema de archivos distribuido que replica y distribuye los datos en múltiples servidores. Luego, introdujeron MapReduce, un modelo de programación distribuido para procesamiento paralelo.
- Impacto: Permitió a Google escalar su infraestructura a millones de peticiones por segundo y procesar petabytes de dato

2. Netflix – Microservicios y distribución global

- Problema: Necesitaban entregar contenido en streaming a millones de usuarios en todo el mundo con alta disponibilidad.
- Solución distribuida: Migraron a una arquitectura de microservicios distribuidos desplegados sobre AWS. Utilizan técnicas de balanceo de carga, replicación y recuperación ante fallos.
- Impacto: Ofrecen una experiencia fluida y resiliente incluso ante fallos parciales o picos de tráfico.

3. **Amazon – Comercio electrónico y AWS**

- Problema: Requerían una infraestructura que pudiera soportar altísimos volúmenes de tráfico, como en eventos como el “Prime Day”.
- Solución distribuida: Implementaron una arquitectura distribuida sobre su propia plataforma de nube (AWS), basada en servicios autónomos.

- Impacto: Alta escalabilidad y disponibilidad. AWS se convirtió en el mayor proveedor de servicios en la nube del mundo

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

- Problema: Facebook necesitaba un sistema de base de datos que soportará billones de conexiones y relaciones de usuarios.

- Solución distribuida: Desarrollaron TAO, un sistema de almacenamiento y caché distribuido optimizado para redes sociales.

- Impacto: Millones de usuarios pueden interactuar en tiempo real con actualizaciones consistentes y rápidas.

5. Uber – Plataforma de movilidad en tiempo real

- Problema: Coordinación en tiempo real entre pasajeros y conductores en distintas ciudades del mundo.

- Solución distribuida: Arquitectura basada en servicios distribuidos y comunicación asincrónica, con almacenamiento de estado geolocalizado.

- Impacto: Alta disponibilidad y baja latencia en servicios de transporte, incluso en ciudades con alta densidad.

6. Spotify – Streaming musical basado en servicios distribuidos

- Problema: Proveer streaming musical a millones de usuarios simultáneos, gestionando catálogos, usuarios, playlists y recomendaciones.

- Solución distribuida: Arquitectura de microservicios desplegada en contenedores y balanceada geográficamente.

- Impacto: Escalabilidad global y personalización dinámica para cada usuario.

Herramientas y Tecnologías en los Sistemas distribuidos

En los sistemas distribuidos, existen muchas herramientas y tecnologías que facilitan la comunicación, coordinación, sincronización, monitoreo y gestión de los diferentes componentes distribuidos. A continuación se presentara una clasificación organizada por categorías, con ejemplos actuales y ampliamente usados:

1. **Comunicación y Mensajería**
    - HTTP/REST, gRPC – Protocolos de comunicación entre servicios.
    - RabbitMQ, Apache Kafka – Sistemas de mensajería (colas y streaming de eventos).
    - ZeroMQ, NATS – Middleware ligero de mensajería.

2. **Orquestación y Contenedores:** Facilitan el despliegue, escalado y gestión de aplicaciones distribuidas.
    - Docker – Contenedores ligeros para empaquetar servicios.
    - Docker Compose – Define y ejecuta múltiples contenedores localmente.
    - Kubernetes – Orquestación de contenedores a gran escala.
    - Helm – Gestor de paquetes para Kubernetes.

3. **Bases de Datos Distribuidas:** Almacenan y replican datos entre nodos.
    - MongoDB, Cassandra – NoSQL distribuidas.
    - CockroachDB, YugabyteDB – SQL distribuidas y tolerantes a fallos.
    - Redis (Cluster Mode) – Base de datos en memoria distribuida.
4. **Sistemas de Archivos Distribuidos:** Permiten almacenar archivos accesibles desde varios nodos.
    - HDFS (Hadoop Distributed File System)
    - Ceph
    - GlusterFS

5. **Monitoreo y Observabilidad:** Ayudan a observar el comportamiento y rendimiento del sistema distribuido.
    - Prometheus – Recolección de métricas.
    - Grafana – Visualización de métricas.
    - Loki + Promtail – Recolección y visualización de logs.
    - Jaeger, Zipkin – Trazabilidad distribuida (distributed tracing)

6. **Balanceadores de Carga y Proxies:** Dirigen el tráfico entre servicios distribuidos.
    - NGINX, HAProxy – Balanceadores de carga clásicos.
    - Envoy, Traefik – Proxies modernos compatibles con microservicios.
    - Kong API Gateway – Control de tráfico de APIs distribuidas.

7. **Sincronización y Coordinación:** Coordinan procesos distribuidos.
    - Apache ZooKeeper – Coordinación y sincronización.
    - etcd – Usado también para mantener consistencia de configuración en clústeres (como Kubernetes).

8. **Frameworks y Plataformas de Desarrollo:** Facilitan la construcción de sistemas distribuidos.
    - Spring Cloud (Java) – Framework para microservicios.
    - Akka (Scala/Java) – Para sistemas reactivos distribuidos.
    - Apache Spark – Procesamiento de datos distribuido.

9. **Descubrimiento de Servicios:** Permiten que los servicios se encuentren entre sí.
    - Cónsul – Descubrimiento y configuración de servicios.
    - Eureka (de Netflix OSS) – Muy usado en entornos Spring.
    - etcd – Almacenamiento clave-valor distribuido para configuración.

Retos en Sistemas Distribuidos

Los sistemas distribuidos, si bien ofrecen muchas ventajas como escalabilidad, tolerancia a fallos y disponibilidad, también presentan algunas dificultades en áreas técnicas y conceptuales sobre todo diseño, implementación y operación.

1. **Consistencia de Datos:** Cuando múltiples nodos acceden y modifican datos simultáneamente, mantener la coherencia es complejo.
    - Problema: ¿Cómo asegurar que todos los nodos vean el mismo estado del sistema?
    - Ejemplo: Conflictos de escritura en bases de datos distribuidas.
    - Soluciones: Algoritmos de consenso (Raft, Paxos), sistemas con consistencia eventual.

2. **Sincronización de Relojes:** Los relojes en diferentes máquinas pueden estar desincronizados.
    - Problema: Los eventos ocurren en diferentes momentos según cada reloj local.
    - Ejemplo: Una transacción parece ocurrir “antes” de otra que en realidad fue anterior.
    - Soluciones: Protocolos como NTP, relojes lógicos (Lamport), relojes vectoriales.

3. **Fallos de Red y Comunicación:** La red no es confiable; pueden perderse mensajes, duplicarse o llegar tarde.
    - Problema: ¿El nodo no responde porque está caído o por un retraso en la red?
    - Ejemplo: Timeout en un microservicio por latencia alta.
    - Soluciones: Retransmisión, timeouts inteligentes, circuit breakers.

4. **Heterogeneidad y Portabilidad:** Los nodos pueden tener hardware, OS o lenguajes distintos.
    - Problema: Cómo hacer que trabajen juntos de manera eficiente.
    - Ejemplo: Un microservicio en Java se comunica con otro en Python.
    - Soluciones: Estandarización (REST/gRPC), contenedores (Docker).

5. **Actualizaciones y Mantenimiento:** Actualizar un sistema distribuido sin interrumpir el servicio es complejo.
    - Problema: Evitar interrupciones al hacer mantenimiento.
    - Ejemplo: Incompatibilidad entre versiones de servicios.
    - Soluciones: Despliegues blue/green, rolling updates, versionado de APIs.

6. **Balanceo de Carga y Escalabilidad:** Repartir correctamente la carga entre múltiples nodos para evitar cuellos de botella.
    - Problema: Algunos nodos reciben más carga que otros.
    - Ejemplo: Una API Gateway sobrecargada.
    - Soluciones: Balanceadores de carga (NGINX, Envoy), autoscaling, replicación.

7. **Seguridad Distribuida:** Mantener actualizada y consistente la configuración distribuida es difícil.
    - Problema: ¿Cómo encontrar y configurar servicios que cambian de dirección IP?
    - Ejemplo: Un contenedor cambia de host y otros servicios no lo encuentran.
    - Soluciones: Consul, etcd, Eureka para descubrimiento de servicios.

Tópicos Avanzados

1. Consenso Distribuido (Raft, Paxos):

Es el proceso mediante el cual múltiples nodos acuerdan un único valor o estado, a pesar de fallos.Es importante por que es la base para mantener consistencia en sistemas distribuidos (por ejemplo, en bases de datos replicadas, líderes en clústeres, etc.).

Algoritmos:

- **Paxos:** Teóricamente sólido pero difícil de implementar.
- **Raft:** Más comprensible y práctico, usado en etcd, Consul, etc.

Aplicaciones:

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

Tecnologías:

- etcd (Raft), Consul, ZooKeeper (Zab: similar a Paxos).

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

Es una arquitectura en la que los componentes se comunican a través de eventos asíncronos en lugar de llamadas directas.

Ventajas:

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

Componentes comunes:

- Productores: Emite eventos (ej: usuario realiza una compra).
- Broker de mensajes: Transporta eventos (ej: Kafka, RabbitMQ).
- Consumidores: Reaccionan a eventos (ej: facturación, inventario)

Tecnologías:

- Apache Kafka, RabbitMQ, NATS, AWS SNS/SQS, Google Pub/Sub
- Frameworks: Axon, Spring Cloud Stream, Kafka Streams

3. **Computación Sin Servidor (Serverless)**

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

Características:

- Escala automática.
- Se paga solo por el tiempo de ejecución.
- Ideal para tareas pequeñas, eventos o APIs.

Casos de uso:

- Procesamiento de eventos (event-driven computing).
- Automatización backend.
- Microservicios altamente desacoplados.

Plataformas populares:

- AWS Lambda, Azure Functions, Google Cloud Functions
- Frameworks: Serverless Framework, OpenFaaS, Knative

Retos:

- Cold start (inicio lento).
- Debugging y testing más complejos.
- Limitaciones en tiempo y memoria.

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

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

- Consistencia: Todos los nodos ven la misma información al mismo tiempo.
- Availability: Cada solicitud recibe una respuesta (exitosa o fallida).
- Partition Tolerance: El sistema sigue funcionando incluso si hay fallos de red.

Implicación:
En presencia de partición (inevitable en redes), debes elegir entre:

- CP: Mantener consistencia sacrificando disponibilidad. Ej: Zookeeper, etcd.
- AP: Mantener disponibilidad sacrificando consistencia. Ej: Cassandra, DynamoDB.
- CA: Sólo posible si no hay particiones (no realista en sistemas distribuidos).

Zero MQ

Es una librería de mensajería asincrónica de alto rendimiento que permite la comunicación entre procesos (IPC), hilos, y sistemas distribuidos. Funciona como una capa de mensajería que se sitúa sobre protocolos de red como TCP, IPC o multicast, permitiendo a los desarrolladores crear sistemas distribuidos y paralelos de manera sencilla y eficiente.

Características

1. **Mensajería asíncrona:**

   ZeroMQ permite a los componentes de una aplicación comunicarse de forma asíncrona, lo que significa que no necesitan esperar a que la otra parte esté lista para responder.

2. **Sin intermediario:**

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

3. **Soportes para diversos patrones de comunicación:**

   ZeroMQ proporciona distintos tipos de sockets y patrones de comunicación, como "pub-sub", "request-reply" y "distribución de tareas", que facilitan la interacción entre los componentes de una aplicación.

4. **Versatilidad:**

   ZeroMQ se puede utilizar en una amplia variedad de contextos, desde sistemas de procesamiento de datos en tiempo real hasta aplicaciones de red, gracias a su eficiencia y flexibilidad.

5. **Disponibilidad multiplataforma:**

   ZeroMQ está disponible en múltiples lenguajes de programación, lo que facilita su integración en diferentes entornos de desarrollo.

6. **Eficiencia:**

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

Historia y evolución

ZeroMQ nació a partir de la necesidad de contar con una herramienta de mensajería ligera, rápida y sin intermediarios, que facilitara el desarrollo de sistemas distribuidos modernos sin la sobrecarga típica de los brokers tradicionales (como JMS o AMQP). Fue desarrollado inicialmente por iMatix Corporation, una compañía conocida por su trabajo en protocolos de mensajería como AMQP (Advanced Message Queuing Protocol), bajo el liderazgo de Pieter Hintjens, un influyente ingeniero de software y defensor del software libre.

ZeroMQ pasó de ser una librería a un ecosistema. Aunque comenzó como una librería C++, hoy existen bindings oficiales y no oficiales para más de 40 lenguajes de programación, además de que realizó muchos cambios de filosofía pues a diferencia de los sistemas con brokers, ZeroMQ sigue el enfoque de “smart endpoints, dumb pipes” (los extremos son inteligentes, la red no) y finalmente en los últimos años ha mejorado altamente su eficiencia dado que utiliza técnicas de bajo nivel como zero-copy, colas internas de mensajes, y comunicación no bloqueante.

Evolución:

- 2007: Se inician los primeros desarrollos internos del proyecto en iMatix.
- 2008: Se libera la primera versión pública de ZeroMQ (v0.x).
- 2009–2010: Se gana popularidad en comunidades de sistemas distribuidos y trading financiero por su velocidad.
- 2010: Publicación de ZeroMQ v2.0, con mejoras en estabilidad, soporte multilenguaje y patrones de mensajería.
- 2011: Se forma la ZeroMQ Community, impulsando un desarrollo más abierto y mantenido por voluntarios.
- 2013: Pieter Hintjens publica el libro "ZeroMQ: Messaging for Many Applications", consolidando la filosofía y diseño del proyecto.
- 2014–2018: Se estabiliza la versión v4.x, agregando más protocolos de transporte, seguridad (CURVE), y soporte para IPv6.
- 2020–presente: ZeroMQ sigue activo con soporte multiplataforma, integración con lenguajes como C, C++, Python, Go, Java, Rust, y soporte para entornos embebidos y IoT.

Ventajas y desventajas

Ventajas	Desventajas
Alto rendimiento: Usa técnicas como *zero-copy* y *asynchronous I/O*, logrando muy baja latencia.	Sin persistencia integrada: A diferencia de RabbitMQ o Kafka, no almacena mensajes si un receptor está desconectado.
Sin necesidad de broker: Comunicación directa entre procesos, reduciendo puntos de fallo.	Manejo de errores manual: El desarrollador debe implementar lógica para reintentos y recuperación.
Multiplataforma y multilenguaje: Disponible para C, C++, Python, Java, Go, Rust, entre otros.	Curva de aprendizaje: Aunque es flexible, entender sus patrones y construir lógica robusta lleva tiempo.
Patrones de comunicación integrados: Soporta request-reply, pub-sub, pipeline, entre otros.	No incluye autenticación avanzada por defecto: Seguridad como CURVE debe ser configurada manualmente.
Ligero y embebible: Ideal para sistemas con recursos limitados (IoT, edge computing).	No tiene monitoreo o administración web: A diferencia de soluciones como RabbitMQ o Kafka.
Extensible y modular: Se puede integrar fácilmente con otras bibliotecas o frameworks.	Requiere sincronización externa: No maneja control central de flujo o orquestación de mensajes.

Casos de uso (Situaciones y/o problemas donde se pueden aplicar)

ZeroMQ es una herramienta poderosa en entornos donde la comunicación es rápida, distribuida y flexible entre procesos o sistemas. A continuación presentaré diversas situaciones en las que ZeroMQ puede aportar múltiples ventajas:

1**. Sistemas distribuidos sin intermediarios**

Uno de los mayores problemas en algunas industrias es la necesidad de comunicación entre múltiples servicios sin depender de un broker central que pueda convertirse en un cuello de botella, para este problema ZeroMQ permite que los nodos se comuniquen directamente entre sí, aumentando la resiliencia y reduciendo la latencia.

2**. Procesamiento de datos en tiempo real**

Un problema frecuente radica a la hora de procesar grandes volúmenes de datos a medida que llegan, como en sistemas financieros o monitoreo de sensores. ZeroMQ maneja una baja latencia y alto rendimiento, lo que permite flujos de datos continuos sin interrupciones.

3. **Comunicación entre hilos o procesos dentro de una misma máquina**

A la hora de compartir datos entre componentes de una aplicación que corren en distintos hilos o procesos (por ejemplo, en sistemas embebidos o microcontroladores) ZeroMQ es bastante útil con su transporte inproc o ipc, el cual permite una comunicación local eficiente y sin dependencias externas.

4. **Sistemas de publicación/suscripción (pub-sub)**

A la hora de enviar mensajes desde una fuente a múltiples receptores interesados (por ejemplo, para notificaciones o eventos). ZeroMQ es muy util por su patrón pub-sub permite una distribución eficiente y desacoplada de mensajes a múltiples suscriptores.

5**. Desarrollo de microservicios o arquitecturas orientadas a servicios (SOA)**

Cuando es necesario coordinar múltiples servicios independientes que deben comunicarse de forma eficiente, para esto ZeroMQ ofrece una alternativa ligera a HTTP/REST o gRPC, ideal para entornos que requieren menor latencia o overhead.

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

Para Sincronizar sensores, controladores y unidades de procesamiento distribuidas ZeroMQ tiene la capacidad de trabajar en redes locales o dispositivos embebidos lo que lo hace ideal para soluciones IoT o tiempo real.

7. Balanceo de carga entre trabajadores

ZeroMQ es muy útil para distribuir tareas entre múltiples "workers" o procesos que deben ejecutarlas de forma concurrente, dado que con el patrón pipeline, permite distribuir trabajos a múltiples nodos balanceando la carga de manera automática.

Casos de aplicación (Ejemplos y casos de éxito en la industria)

ZeroMQ ha sido adoptado en múltiples industrias gracias a su bajo consumo de recursos, alta velocidad y flexibilidad para construir sistemas distribuidos. A continuación, presentaré algunos casos de éxito reales donde ZeroMQ ha sido utilizado como núcleo de comunicación eficiente.

1. **CERN – Control de aceleradores de partículas**

- Aplicación: ZeroMQ se utiliza para la orquestación y comunicación entre sistemas distribuidos de control en los aceleradores de partículas, como el Gran Colisionador de Hadrones (LHC).

- Impacto: Ofrece baja latencia y alta confiabilidad para sistemas críticos en tiempo real, reemplazando tecnologías más complejas y pesadas como CORBA.

2. **NASA – Prototipos de robótica y sistemas embebidos**

- Aplicación: En entornos de simulación y control robótico, ZeroMQ se ha utilizado para probar arquitecturas de comunicación descentralizadas en misiones espaciales y vehículos autónomos.

- Impacto: Facilita prototipado rápido y comunicación entre módulos de software en entornos altamente restringidos.

3. **Spotify – Prototipado de pipelines de procesamiento**

- Aplicación: Aunque no es su herramienta principal de mensajería, Spotify ha usado ZeroMQ en algunos prototipos y pruebas de concepto para manejar flujos de datos de eventos antes de pasar a soluciones definitivas como Kafka.

- Impacto: Permite desarrollar e iterar rápidamente nuevas ideas gracias a su simplicidad y rendimiento

4. Ventas de alta frecuencia – Fintech

- Aplicación: ZeroMQ es muy popular en plataformas de trading de alta frecuencia, donde el rendimiento y la baja latencia son críticos.

- Empresas como: MetaTrader, TickZoom, y varias firmas privadas lo utilizan como mecanismo base de mensajería entre módulos de trading, análisis y ejecución.

- Impacto: Ofrece procesamiento en milisegundos o microsegundos sin necesidad de brokers intermedios.

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: código, librerías, dependencias, configuraciones, etc.Esto asegura que la aplicación se ejecute de la misma manera sin importar en qué computadora o servidor esté corriendo.

Docker sirve para empaquetar aplicaciones con todas sus dependencias, evitar errores por diferencias entre entornos (por ejemplo, "en mi máquina sí funciona"), desplegar fácilmente aplicaciones en distintos entornos (desarrollo, pruebas, producción) y para ahorrar recursos, ya que los contenedores comparten el sistema operativo.

Características

Docker se ha convertido en una herramienta esencial para el desarrollo y despliegue de aplicaciones modernas. Sus características permiten crear entornos ligeros, portables y consistentes, facilitando la ejecución de aplicaciones en cualquier sistema sin importar las diferencias del entorno, sus principales características son:

1. **Contenedores ligeros**

Usan el mismo kernel del sistema operativo (a diferencia de las máquinas virtuales que necesitan un sistema completo).Esto los hace más rápidos y con menor consumo de recursos.

2. **Portabilidad**

Un contenedor Docker se puede ejecutar en cualquier sistema que tenga Docker instalado, sin importar si es Linux, Windows o macOS. “Lo que funciona en mi máquina, funcionará en producción”.

3. **Aislamiento**

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

4. **Imágenes reutilizables**

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

5. **Versionado**

Las imágenes tienen versiones, por lo que es posible controlar qué versión exacta de una aplicación está usando el usuario.

6. **Escalabilidad**

Es posible ejecutar múltiples contenedores simultáneamente, lo cual es ideal para microservicios, además de que se integra fácilmente con herramientas como Kubernetes para escalar en producción.

7. **Integración con CI/CD**

Docker se integra perfectamente con pipelines de Integración Continua y Entrega Continua, facilitando los despliegues automáticos.

8. **Redes y volúmenes personalizables**

Es posible configurar redes internas entre contenedores y crear volúmenes para guardar datos que sobrevivan aunque el contenedor se borre.

9. **Docker Compose**

Permite definir y correr múltiples contenedores a la vez con un solo archivo (docker-compose.yml)., este es ideal para aplicaciones complejas que tienen varios servicios (por ejemplo, backend, frontend y base de datos).

Historia y evolución

Docker fue creado en 2013 por la empresa dotCloud como una solución para empaquetar y ejecutar aplicaciones de forma ligera y aislada. Basado en tecnologías de contenedores de Linux, revolucionó el desarrollo de software al facilitar la portabilidad, escalabilidad y despliegue automatizado de aplicaciones. Desde entonces, ha evolucionado hasta convertirse en una herramienta clave en entornos DevOps y de microservicios.

Evolución:

- **Año 2013:**

  Docker fue lanzado por la empresa dotCloud (hoy llamada Docker Inc.). Nació como una herramienta para facilitar la entrega de software mediante contenedores. La principal inspiración de Docker fue en base a tecnologías de Linux como cgroups y namespaces, que permiten crear entornos aislados.

- **Año 2014**

  Docker gana popularidad rápidamente por su facilidad de uso frente a las máquinas virtuales. Se lanza Docker Hub, un repositorio de imágenes públicas.

- **Año 2015**

  Nace la Open Container Initiative (OCI) para estandarizar los contenedores, con apoyo de empresas como Google, Microsoft y Red Hat.

- **2016 en adelante**

  En este año Docker se integra con herramientas de orquestación como Kubernetes. y aparecen herramientas complementarias como Docker Compose, Docker Swarm y mejoras en la gestión de imágenes y seguridad.

  Hoy en día Docker es una de las tecnologías más utilizadas en DevOps, desarrollo de microservicios y despliegues en la nube, con una comunidad global muy activa.

Ventajas y desventajas

Ventajas	Desventajas
Ahorro de tiempo en el desarrollo: Permite montar entornos en segundos, evitando configuraciones manuales repetitivas.	**Menor aislamiento que una VM:** Comparte el kernel del sistema operativo, lo que puede representar un riesgo de seguridad en entornos sensibles.
Facilita el trabajo en equipo: todos los desarrolladores pueden usar el mismo entorno sin inconsistencias.	**Curva de aprendizaje inicial:** Requiere entender conceptos como imágenes, volúmenes, redes y orquestadores.
Mejora la calidad del software: Reduce los errores por diferencias de entorno, lo que ayuda a detectar bugs reales más rápido.	**Gestión de datos persistentes:** Al borrar un contenedor se pierden los datos, a menos que se usen volúmenes correctamente.
Despliegues más seguros y controlados: Puedes probar una imagen en staging y luego desplegar exactamente la misma en producción.	**No apto para todas las aplicaciones:** Algunas apps que dependen mucho del sistema operativo pueden no funcionar bien en contenedores
Fomenta la automatización: Se integra fácilmente con herramientas CI/CD, lo que ayuda a automatizar pruebas y despliegues.	**Problemas de seguridad:** la falta de segmentación significa que varios contenedores pueden ser vulnerables a ataques al sistema host.

Casos de uso (Situaciones y/o problemas donde se pueden aplicar)

1**. Desarrollo en equipo**

Cada desarrollador tiene una configuración diferente en su máquina. Docker es muy útil en entornos en los que todos trabajan con el mismo contenedor, evitando el clásico “en mi máquina sí funciona”.

2. **Pruebas automatizadas (CI/CD)**

A la hora de ejecutar pruebas en un entorno limpio, cada vez que se hace un cambio, Docker crea un entorno desde cero para cada ejecución, asegurando que las pruebas sean confiables.

3. **Despliegue en múltiples entornos**

Cuando una aplicación funciona en desarrollo, pero falla en producción, Docker permite desplegar la misma imagen (contenedor) en todos los entornos, asegurando coherencia.

4. **Microservicios**

A la hora de construir una aplicación distribuida con muchos servicios independientes Docker permite que cada microservicio corra en su propio contenedor, facilitando su desarrollo, prueba y despliegue.

5. **Pruebas de versiones o tecnologías**

A la hora de probar una app en diferentes versiones de una base de datos o lenguaje, Docker permite levantar rápidamente distintos contenedores con versiones distintas sin afectar el sistema.

6. **Simulación de entornos complejos**

Si es Necesario levantar una aplicación con varios componentes (base de datos, backend, frontend, cache, etc.), Docker Compose con un solo archivo puedes orquestar todos los contenedores y correr todo en segundos.

Casos de aplicación (Ejemplos y casos de éxito en la industria)

Desde su creación, Docker ha transformado la manera en que las empresas desarrollan, prueban y despliegan software. Grandes compañías de diferentes sectores han adoptado Docker para mejorar la eficiencia, reducir costos y acelerar sus ciclos de desarrollo. A continuación, se presentan algunos ejemplos reales de cómo Docker ha sido implementado con éxito en la industria.

1. **Spotify**

    - Uso: Automatiza pruebas y despliegue de sus servicios musicales.
    - Beneficio: Reducción del tiempo de entrega de nuevas funciones gracias al uso de contenedores y microservicios.

2. **Netflix**
    - Uso: Ejecuta pruebas A/B, microservicios y entornos de prueba con Docker.
    - Beneficio: Mayor velocidad en el desarrollo y despliegue continuo a escala global.

3. **Paypal**
    - Uso: Migró muchas aplicaciones a contenedores Docker.
    - Beneficio: Redujo significativamente el tiempo de puesta en marcha de entornos de desarrollo y producción.

4. **ADP (Automatic Data Processing)**
    - Uso: Usa Docker para entregar rápidamente aplicaciones financieras a sus clientes.
    - Beneficio: Mejoró la consistencia y seguridad del entorno de entrega.
5. **BBC**
    - Uso: Utiliza Docker para la distribución de contenidos digitales y entornos de pruebas.
    - Beneficio: Mayor agilidad en el desarrollo de plataformas multimedia.
6. **eBay**
    - Uso: Desarrolla y despliega microservicios usando Docker.
    - Beneficio: Despliegue más rápido de servicios nuevos sin afectar los sistemas existentes.

Que tan comun es el Stack

1. **ZeroMQ:** es moderadamente común en la industria, especialmente en contextos donde se requiere:
    - Alta velocidad de comunicación entre procesos.
    - Bajo uso de recursos.
    - Comunicación descentralizada sin un broker intermediario.

Aunque no es tan masivo como Kafka, RabbitMQ o MQTT, sigue siendo ampliamente utilizado en nichos específicos donde su rendimiento, simplicidad y flexibilidad ofrecen ventajas importantes.

Ámbitos donde más se utiliza:

Ámbitos	Nivel de uso	Justificación
Finanzas y trading	Alto	Uso intensivo en sistemas de trading algorítmico por su baja latencia.
Sistemas embebidos / IoT	Medio–Alto	Ligereza y compatibilidad con plataformas pequeñas como Raspberry Pi o microcontroladores.
Robótica y automatización	Medio	Funciona para middleware personalizado, con bajo overhead y comunicación local rápida.
Ciencia e investigación	Medio	Adoptado en entornos académicos (CERN, NASA, universidades) para prototipado distribuido.
Web y microservicios (empresas SaaS)	Bajo–Medio	Menor popularidad frente a herramientas más completas como Kafka o RabbitMQ.

Tendencias en el mercado según stack overflow
Las tecnologías han evolucionado a lo largo del tiempo según el uso de sus etiquetas desde 2010 hasta la actualidad (2024)

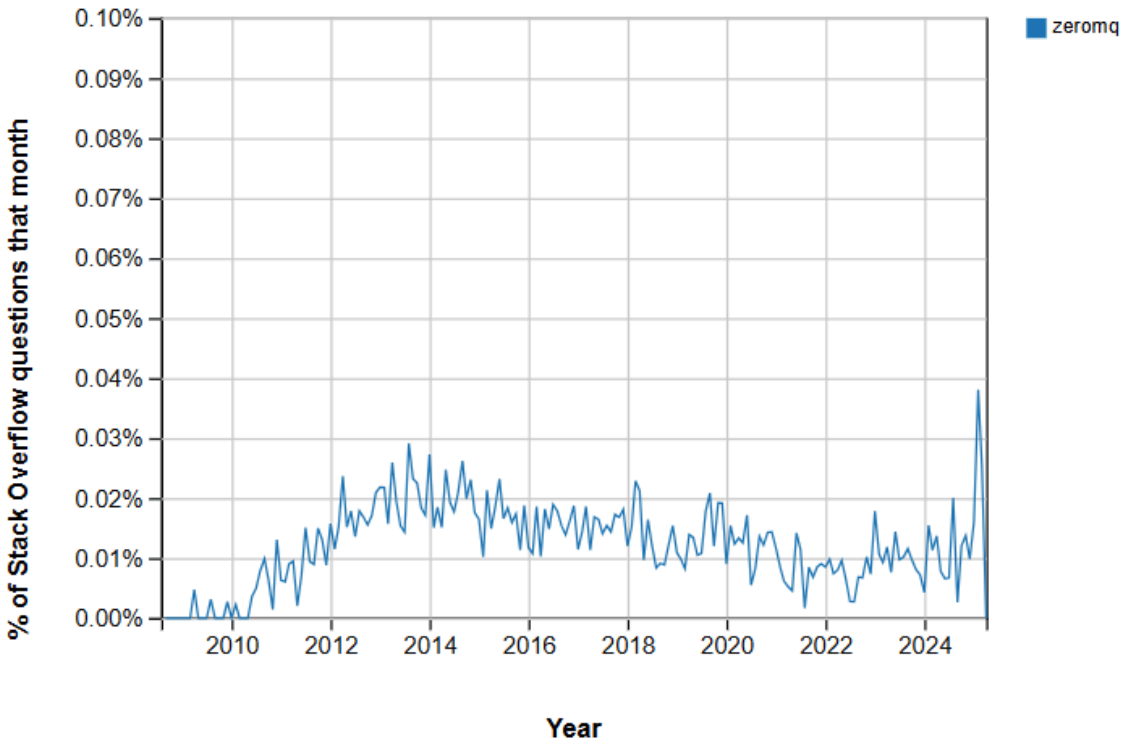


Imagen 1. Tendencias en el mercado - Fuente: stack overflow

Pico inicial de popularidad (2012–2015):

- ZeroMQ alcanzó su mayor nivel de uso relativo en Stack Overflow entre 2012 y 2015.
- Durante este período, llegó a representar entre **0.02% y 0.03%** de todas las preguntas mensuales en la plataforma, lo que indica un interés técnico considerable.

Tendenci baja desde 2016:

- Después de 2015, el porcentaje de preguntas comenzó a disminuir gradualmente, situándose entre **0.005% y 0.015%** en la mayoría de los meses.
- Esto sugiere que, aunque sigue usándose, el interés o adopción general en la comunidad ha disminuido.

Pico anómalo reciente (~2024–2025):

- Se observa un pico abrupto alrededor de 2024-2025.
- Esto podría deberse a un evento puntual: una nueva versión, vulnerabilidad, integración popular o una tendencia temporal en foros o cursos.

Comparación general

- Incluso en su punto más alto, ZeroMQ fue una tecnología de nicho comparada con otras más ampliamente usadas (como reactjs, python, docker, etc.).

Algunos motivos por los que no tiene mayor reconocimiento es por que no tiene broker ni persistencia por defecto, (lo cual es una ventaja en algunos casos, pero una limitación en otros)y además requiere mayor configuración y responsabilidad por parte del desarrollador para temas como reconexión, persistencia o autenticación, por otro lado no tiene tantas herramientas de administración o monitoreo listas para usar como RabbitMQ o Kafka.

2. **Docker**: es una de las herramientas más utilizadas en DevOps, desarrollo web, microservicios y entornos en la nube, esta herramienta suele estar presente en empresas de todos los tamaños, desde startups hasta gigantes como Google, Netflix y Amazon.

   Según el State of DevOps Report y encuestas de Stack Overflow, Docker ha sido durante años una de las herramientas más populares entre desarrolladores y equipos DevOps.

Ámbitos donde más se utiliza:

Ámbitos	**Nivel de uso**	**Justificación**
Desarrollo de software	Alto	Uso extendido para crear entornos reproducibles, testear y desplegar aplicaciones.
DevOps y CI/CD	Alto	Herramienta clave en la automatización del despliegue y pruebas continuas.
Educación y formación técnica	Medio–Alto	Permite practicar con entornos reales sin afectar al sistema operativo principal.
Ciencia e investigación	Medio	Útil para compartir entornos reproducibles y ejecutar simulaciones aisladas.
Sistemas embebidos / IoT	Bajo-Medio	Limitado por recursos, aunque útil en dispositivos potentes como Raspberry Pi.

Tendencias en el mercado según stack overflow

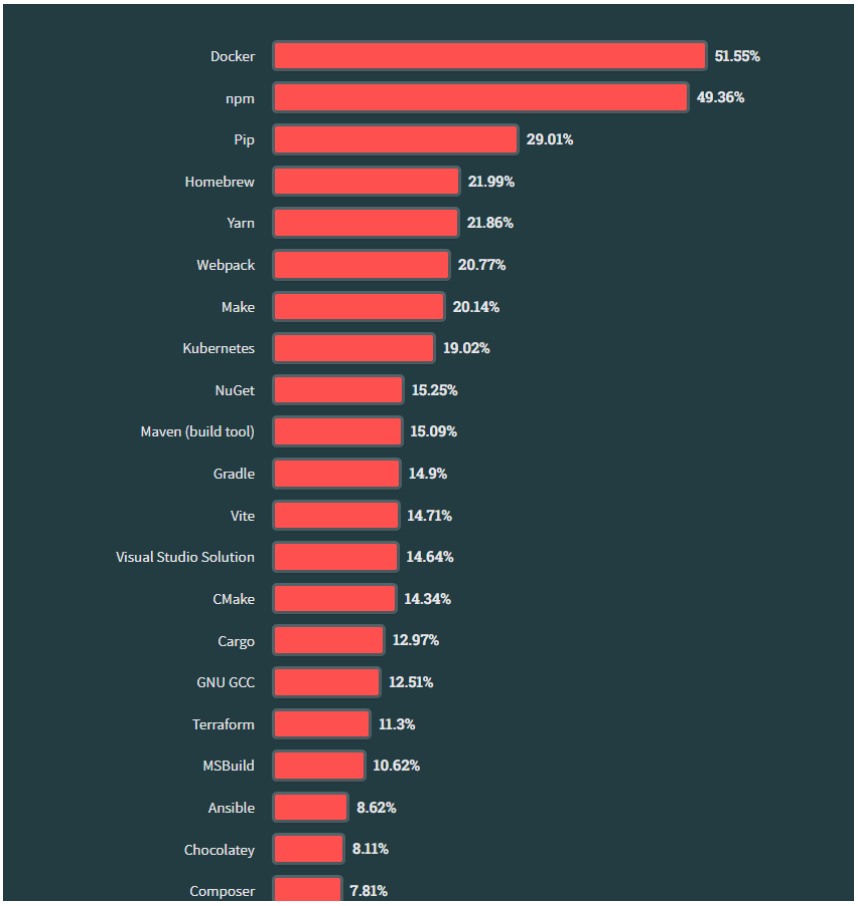


Imagen 2. Tendencias en el mercado - Fuente: stack overflow

Docker mantuvo su posición como la herramienta más utilizada por desarrolladores profesionales, con un 53% de adopción. Esta tendencia resalta la consolidación de Docker como una herramienta esencial en prácticas modernas de DevOps y desarrollo de software.

Matriz de análisis de Principios SOLID vs ZeroMQ

ZeroMQ no es un framework orientado a objetos per se, pero se integra en arquitecturas construidas con lenguajes OO (como Java, C++, C#), y por eso es útil evaluar cómo los principios SOLID influyen en su uso, diseño e integración.

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.

Matriz de análisis de Principios SOLID vs Docker

Docker no implementa directamente los principios SOLID, ya que estos provienen del diseño orientado a objetos, pero su arquitectura y filosofía sí los reflejan en el diseño de sistemas y servicios. Se alinea sobre todo con los principios de responsabilidad única y de inversión de dependencias, claves en arquitecturas modernas como microservicios.

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

Mediante esta matriz se evalúa ZeroMQ con los "atributos no funcionales", los cuales son características que determinan el rendimiento y la utilidad de un sistema más allá de lo que hace funcionalmente.

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.

Acorde a la matriz anterior podemos concluir que ZeroMQ destaca en desempeño, portabilidad y escalabilidad.Sin embargo, atributos como confiabilidad, seguridad y disponibilidad deben ser gestionados por el desarrollador, ya que ZeroMQ delega muchas de esas responsabilidades al diseño del sistema.Por otro lado respecto a Docker este destaca en portabilidad, mantenibilidad y escalabilidad, lo que lo convierte en una herramienta fundamental para el desarrollo moderno y despliegue de aplicaciones. Además, atributos como confiabilidad e interoperabilidad se ven altamente fortalecidos mediante el uso de buenas prácticas y herramientas complementarias como Kubernetes. Sin embargo, aspectos como la seguridad y el desempeño requieren configuraciones adicionales y una adecuada planificación, ya que Docker por sí solo no garantiza protección avanzada ni máximo rendimiento en todos los escenarios.

Matriz de análisis de Tácticas vs ZeroMQ

Para realizar este análisis son mecanismos de diseño que ayudan a alcanzar atributos de calidad como rendimiento, disponibilidad, seguridad, mantenibilidad, etc. Se agrupan según el atributo que soportan.

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.

Mediante la matriz anterior se puede concluir que ZeroMQ soporta muy bien tácticas relacionadas con rendimiento, escalabilidad y comunicación asíncrona.Sin embargo, para tácticas más avanzadas como alta disponibilidad, persistencia y seguridad, se debe implementar componentes externos o añadir lógica personalizada. Adicionalmente no es un message broker como RabbitMQ o Kafka, sino una librería de transporte eficiente, lo cual ofrece flexibilidad pero implica más responsabilidad en el diseño arquitectónico.

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.

Matriz de análisis de Patrones vs ZeroMQ

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 compatibl**e – 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.

En base a la Matriz anterior se observa que ZeroMQ soporta muy bien patrones de mensajería y arquitecturas distribuidas, gracias a su diseño ligero y flexible. Sin embargo no incluye un broker ni persistencia, por lo que patrones como Message Broker o Reliable Message Bus deben implementarse manualmente si se requieren.Por otro lado combinado con patrones como Façade o Adapter, ZeroMQ se integra fácilmente sin acoplarse al sistema principal.

Matriz de análisis de Patrones vs Docker

Esta Matriz evalúa cómo Docker se adapta o soporta distintos patrones arquitectónicos comunes en sistemas distribuidos:

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 Mercado Laboral vs ZeroMQ

Dimensión del Mercado	Análisis respecto a ZeroMQ	Análisis respecto a Docker
Demanda en ofertas laborales	**Baja** – ZeroMQ aparece poco en ofertas laborales en comparación con tecnologías como Kafka o RabbitMQ. Es más común en sectores muy específicos (trading, IoT, sistemas embebidos).	**Alta –** Es una habilidad ampliamente solicitada en DevOps, backend y despliegue de microservicios en múltiples industrias.
Popularidad en comunidades	**Moderada** – Tiene una comunidad activa pero pequeña. En GitHub y StackOverflow tiene preguntas, pero no tanto movimiento como otras tecnologías.	**Muy alta** – Docker tiene una enorme comunidad global con abundante documentación, foros, cursos y soporte continuo.
Curva de aprendizaje	**Accesible** – Su API es clara y bien documentada. Sin embargo, entender los patrones correctamente puede ser un reto sin experiencia previa en sistemas distribuidos.	**Accesible –** Su CLI y concepto de contenedores son fáciles de aprender; además tiene gran documentación y recursos visuales
Casos de uso en la industria	**Alta** – Usado en aplicaciones de alta frecuencia (trading), sistemas en tiempo real, y comunicaciones M2M (IoT). Muy apreciado donde la latencia es crítica.	**Altísima –** Usado por empresas como Netflix, Spotify, PayPal, Airbnb y en cualquier empresa con arquitectura basada en microservicios o CI/CD.
Soporte en lenguajes populares	**Excelente** – Compatible con C, C++, Python, Java, Go, .NET, Rust, entre otros. Muy portable y fácil de integrar.	**Excelente –** Soporta cualquier lenguaje que pueda correr en un contenedor (Python, Java, Go, Node.js, etc.).
Evolución tecnológica	**Estancamiento relativo** – El desarrollo ha disminuido en los últimos años comparado con otras tecnologías emergentes. Aunque estable, su ecosistema crece lentamente.	**Muy activa –** Docker y su ecosistema (Docker Compose, BuildKit, etc.) se actualizan frecuentemente; además, interoperable con Kubernetes y otras herramientas DevOps.
Salarios vinculados	**Baja** – No suele verse como tecnología principal en las ofertas, por lo tanto no se traduce en una ventaja salarial directa.	**Alta –** Tiene Alta demanda profesional incrementa los salarios en roles que incluyen Docker (DevOps, backend, SRE, arquitectos cloud, etc.).

Acorde al diagrama anterior, se puede observar que ZeroMQ es una herramienta poderosa pero no es muy común en empresas mainstream ni en la mayoría de ofertas laborales. Sin embargo, es muy valorada en entornos especializados donde el rendimiento y la latencia son críticos, como en FinTech, IoT o comunicaciones en tiempo real. Aprender ZeroMQ puede otorgar una ventaja técnica en estos contextos específicos, aunque no garantiza un aumento inmediato en la empleabilidad general.

En contraste, Docker goza de una altísima adopción en el mercado laboral actual, siendo una habilidad prácticamente estándar en perfiles de DevOps, backend y arquitecturas modernas basadas en microservicios. Su popularidad, facilidad de aprendizaje y fuerte comunidad lo convierten en una tecnología que sí influye positivamente en la empleabilidad y en las expectativas salariales dentro del sector tecnológico.

Implementación

Esta implementación busca crear un sistema distribuido que ayude a detectar posibles incendios forestales en zonas boscosas, vigilando tres factores importantes: temperatura del aire, humedad y presencia de humo.

El sistema se divide en tres partes:

- Zona de recolección: donde están los sensores. Se simulan sensores de humo, temperatura y humedad que envían datos cada segundo. Si se detecta humo, se activa un aspersor y se manda una alerta.

- Centro de control: recibe los datos de los sensores, los revisa, calcula promedios y manda alertas si hay valores peligrosos. También envía todo a la nube. Esta capa tiene un sistema de respaldo por si el componente principal falla.

- Cloud (nube): guarda los datos y genera reportes mensuales. También recibe y almacena todas las alertas.

Todos los componentes se comunican usando ZeroMQ,dado que esta herramienta es la que permite enviar mensajes entre ellos de forma rápida. Además, se usa Docker para ejecutar cada parte del sistema en contenedores, lo que facilita su uso en diferentes computadores o máquinas virtuales. El programa no solo simula un sistema real de monitoreo, sino que también prueba cómo se comporta ante fallos y qué tan bien se comunica entre sus partes. Esta implementación servirá como prototipo demostrativo para aplicaciones reales como:

- Monitoreo de sistemas industriales.
- Comunicaciones máquina a máquina (M2M).
- Alertas financieras o de seguridad en sistemas críticos.
- Notificaciones de stock en e-commerce.
- Plataformas de trading o simulaciones en entornos fintech.

Diagramas

1. Diagrama de Contexto

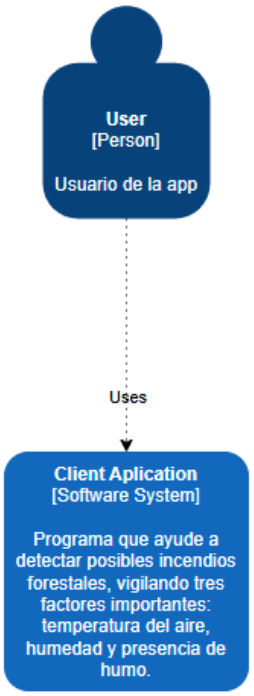


Imagen 3. Diagrama de Contexto

Este diagrama muestra una vista general del sistema y cómo interactúan los usuarios externos con él. Es útil para entender quién utiliza la herramienta y cuál es su propósito principal. En este caso, el sistema está diseñado para que los usuarios puedan realizar simulaciones de incendios forestales mediante distintos sensores (Humo, Humedad, Temperatura, etc)

Componentes

1. **User [Persona]**
   - Es el usuario de la aplicación.
   - Representa a cualquier persona que quiera realizar una simulación.

2. **Client Application [Software System]**
   - Es un programa que permite a los usuarios interactuar con el sistema.
   - El programa sigue una arquitectura distribuida en tres niveles: Zona de recolección → Centro de control → Nube (Cloud).

Flujo:

1. User ejecuta los comandos para iniciar la aplicación
2. Se activan múltiples sensores simulados de temperatura, humedad y humo (10 de cada uno).
3. El proxy envía los datos procesados hacia la nube (nivel Cloud) usando el patrón Request-Reply.
4. El sistema responde y muestra los resultados o confirmaciones.

2. Diagrama de Alto Nivel

Mediante este diagrama de alto nivel se va a representar gráficamente el flujo y arquitectura del sistema. Este diagrama mostrará los componentes principales del programa, cómo se comunican entre sí sus componentes y su distribución lógica.

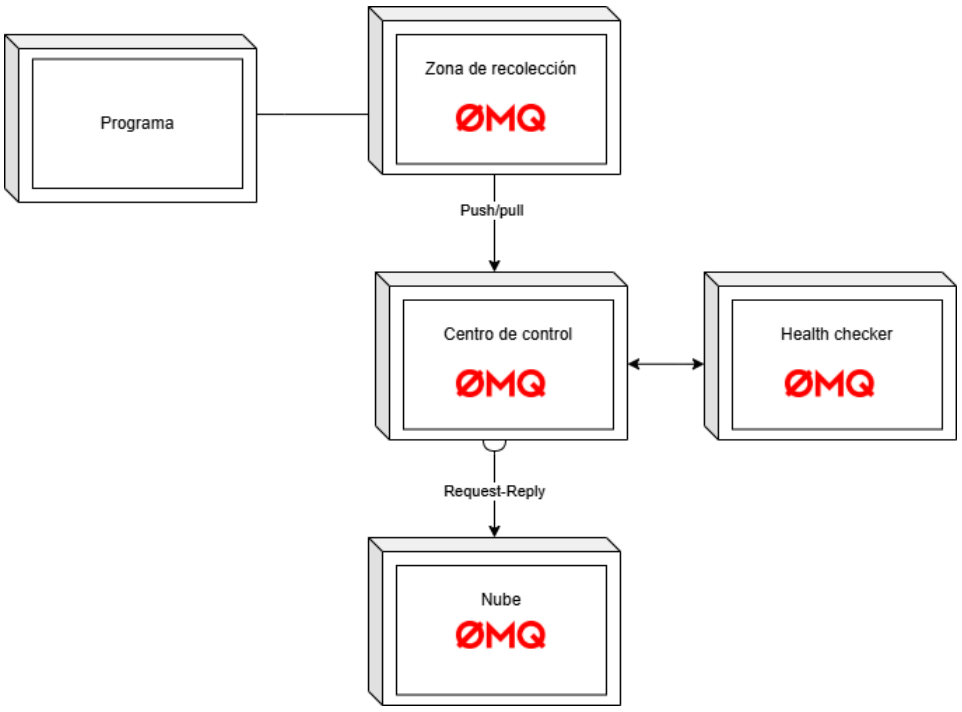


Imagen 4. Diagrama de Alto nivel

Componentes y Flujo

1. Zona de Recolección:

Este componente tiene sensores que envían datos al Proxy local (Centro de control).

2. Centro de Control:

Este componente tiene el proxy el cual recibe, valida y calcula promedios, envía alertas y datos procesados a la nube

3. Health checker:

Es un proxy de respaldo por si el proxy principal falla, este sustituirá todas sus funciones

4. Nube (Cloud)

Es el encargado de almacenar los datos, es decir guarda las alertas, calcula promedios y genera alertas globales, además de tener integrado el sistema de calidad.

Comunicación

- PUSH-PULL (ZeroMQ): los sensores se comunican de manera asíncrona con el proxy (datos) mediante el patrón Push/pull. Los sensores envían datos de manera continua sin esperar respuesta y el proxy simplemente los recibe cuando llegan.
- Request-Reply (ZeroMQ): Entre el proxy y la nube la comunicación es síncrona y el patrón que implementa es Request/Reply. En este caso el emisor espera una respuesta del receptor (por ejemplo, para confirmar que la alerta fue recibida y publicada).

3. Diagrama Dinámico

El diagrama dinámico representa el flujo de comunicación entre los diferentes componentes del sistema en tiempo de ejecución, específicamente durante el proceso de alertas por parte de los sensores. Este diagrama muestra cómo se gestionan las solicitudes desde la zona de recolección hasta el sistema de calidad. A continuación se muestra a detalle todo el flujo del programa.

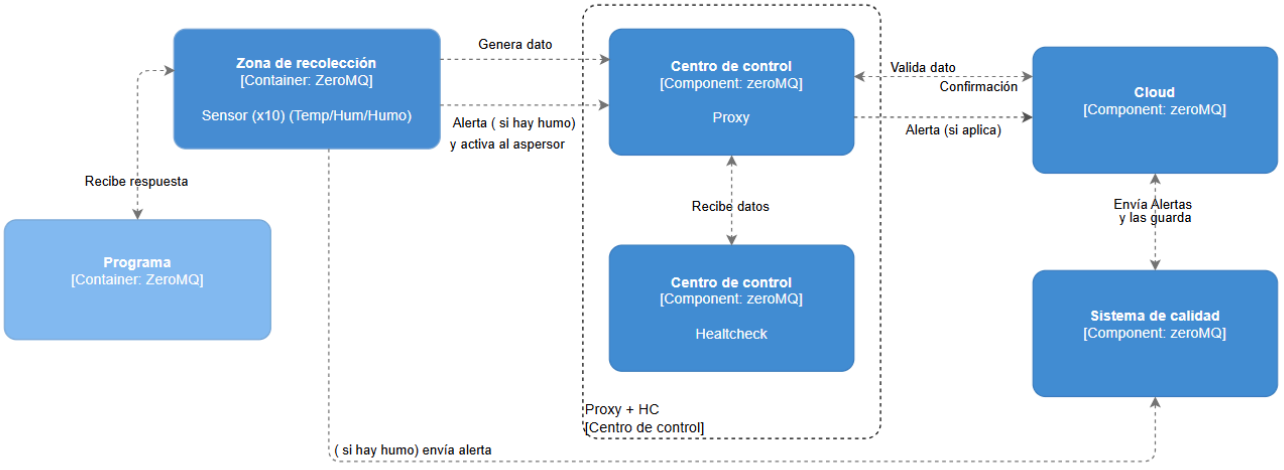


Imagen 5. Diagrama Dinámico C4

Componentes y Flujo:

1. Zona de recolección:
   Los sensores envían datos al proxy del centro de control.
   Si hay humo, además:

   - Activan el aspersor.
   - Envían alerta al sistema de calidad local.

2. Centro de control:
   Tiene el proxy, el cual valida y envía los datos a la nube.

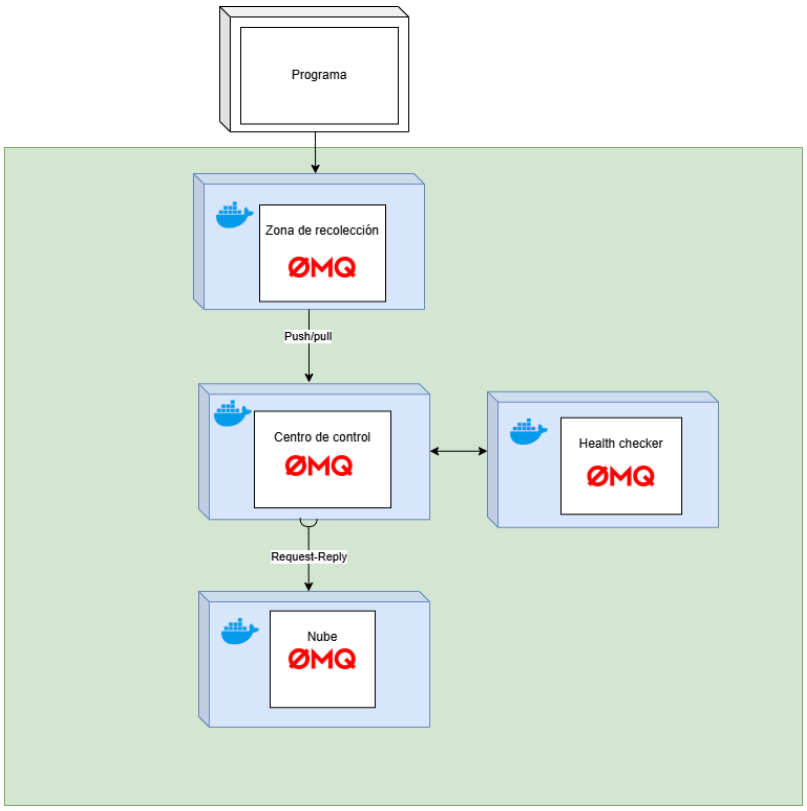
3. Healthcheck
   El proxy tiene tolerancia a fallos: si se cae, el backup asume la carga.

4. Cloud:
   Se generan alertas en distintos niveles según el dato (fuera de rango, promedio, mensual...).

5. Sistema de calidad:
   El sistema de calidad recibe y muestra alertas.

4. Diagrama de Despliegue

El diagrama de despliegue representa la distribución física de los componentes del sistema dentro de contenedores Docker. Muestra cómo están desplegados cada uno de los módulos del programa, en este caso cada componente se ejecuta de forma independiente, facilitando el mantenimiento, la escalabilidad y la administración del sistema



Componentes y Flujo:

1. Inicio del programa

    - El programa principal inicia los contenedores y procesos.
    - Se activan los sensores simulados (zona de recolección).

2. Zona de recolección

    - Contenedor con sensores de humo, temperatura y humedad.
    - Envían datos mediante ZeroMQ (Push) hacia el Centro de control usando el patrón Push/Pull.
    - Si se detecta humo, se pueden generar alertas y activar un actuador (no mostrado aquí).

3. Centro de control

    - Recibe datos de los sensores, valida y procesa.
    - Si detecta datos fuera de rango, genera alertas.
    - Se comunica con la Nube usando ZeroMQ con Request-Reply, enviando resultados procesados o alertas.

4. Nube (Cloud)

    - Almacena las alertas, realiza cálculos mensuales, y guarda datos para análisis posteriores.

5. Health Checker

    - Supervisa que el Centro de control (proxy) esté funcionando correctamente.
    - Si detecta una falla, puede activar un proxy de respaldo
    - La comunicación con el proxy también se da usando ZeroMQ.

    Docker en el sistema: cada componente (zona de recolección, centro de control, nube, health checker) está contenedorizado con Docker, lo que ofrece:

    - Aislamiento entre procesos
    - Facilidad para escalar o reiniciar servicios
    - Red interna definida (con docker-compose)

5. Diagrama de Paquetes UML

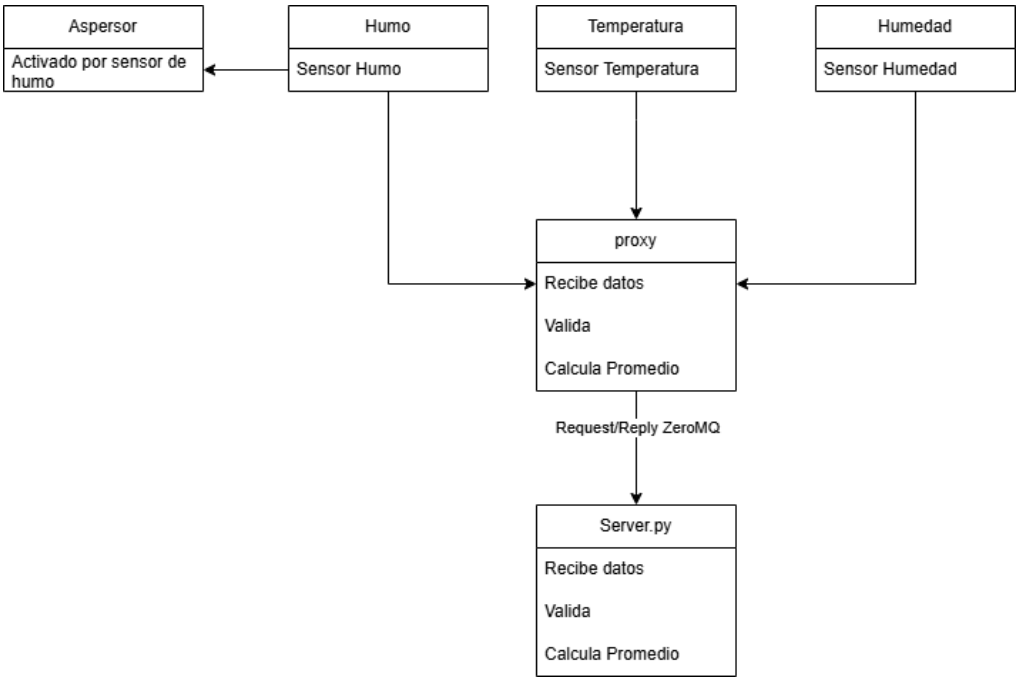


Imagen 6. Diagrama de Paquetes UML C4

La estructura de paquetes del programa es bastante sencilla, dado que únicamente están los componentes principales de la aplicación, ya que el único programa que se utiliza en todos los casos es ZeroMQ, donde cada paquete tiene su código relacionado.

Componentes:

1. Sensores: agrupados en paquetes Humo, Temperatura y Humedad, contienen la lógica para la generación de datos simulados.

2. Proxy (Centro de Control): en el paquete proxy, se centraliza la recepción y validación de datos provenientes de los sensores, así como el cálculo de promedios.

3. Server (Nube): el paquete server gestiona el almacenamiento de datos, cálculos agregados mensuales y emisión de alertas a largo plazo.

4. Aspersor: contenido en su propio paquete, responde ante la activación inmediata desde los sensores de humo.

5. Orquestación: a través del archivo docker-compose.yml, se definen y conectan los contenedores de cada módulo.

6. Diagrama de Componentes C4

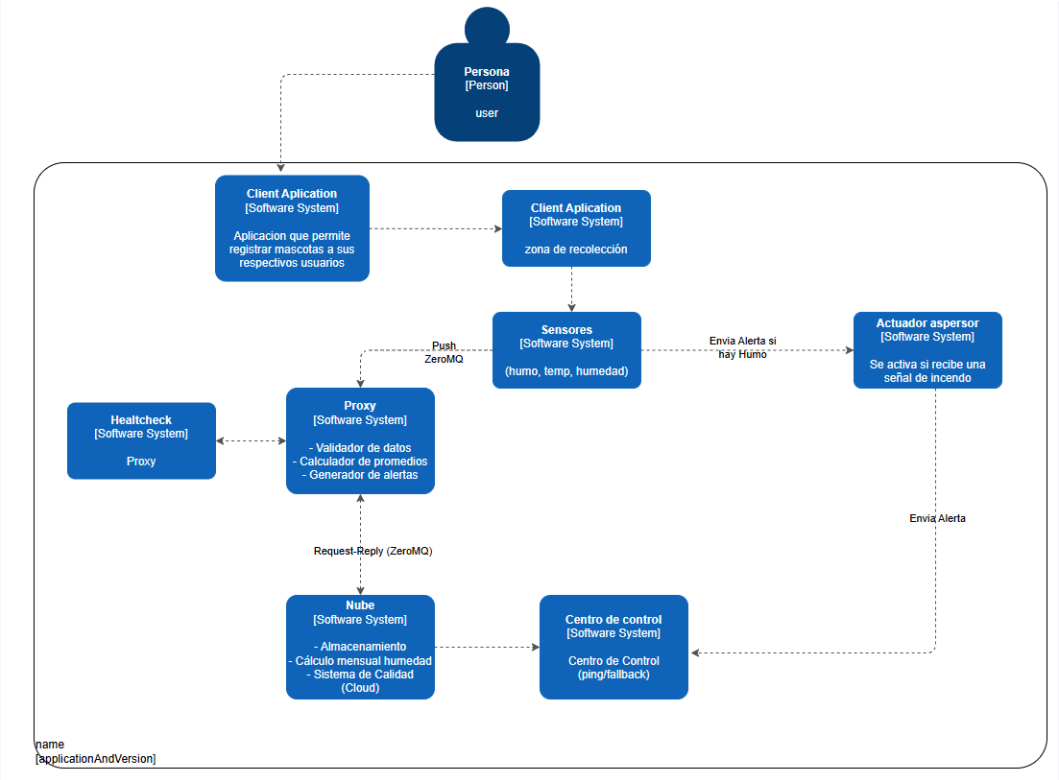


Imagen 7. Diagrama de Componentes C4

Este diagrama de componentes detalla la estructura interna del sistema detector de incendios, enfocándose en los principales componentes de software que interactúan entre sí dentro del contenedor lógico de la aplicación. Refleja la interacción del usuario, y todos los componentes de la aplicación, es decir los sensores, el proxy, la nube y el centro de control

Componentes principales y flujo:

1. Programa principal
    - Orquestador general (puede lanzar los contenedores y coordinar la ejecución).

2. Contenedor: Zona de Recolección
    - Componentes internos:
        - Sensor de humo (x10)
        - Sensor de temperatura (x10)
        - Sensor de humedad (x10)
        - Actuador aspersor: Se activa únicamente si recibe alertas de humo

- Comunicación:
    - PUSH de datos hacia el centro de control
    - Request a SC local (cuando hay alerta de humo)

3. Contenedor: Centro de Control

    - Componentes internos:
        - Receptor PUSH (ZeroMQ)
        - Validador de datos
        - Calculador de promedios
        - Generador de alertas
    - Comunicación:
        - PULL desde sensores
        - Request-Reply hacia la Nube
        - Comunicación bidireccional con el Health Check

4. Contenedor: Health Checker

    Monitoriza la disponibilidad del centro de control, Si hay falla, activa proxy de respaldo

    - Comunicación:
        - Ping / Heartbeat al proxy
        - Comunicación directa con proxy de respaldo

5. Contenedor: Nube
    - Componentes internos:
        - Almacenamiento histórico
        - Cálculo mensual de humedad
        - Sistema de calidad (Cloud)
    - Comunicación:
        - Recepción de datos vía Request-Reply
        - Request a SC Cloud si hay alerta mensual

7. Diagrama de Contenedores

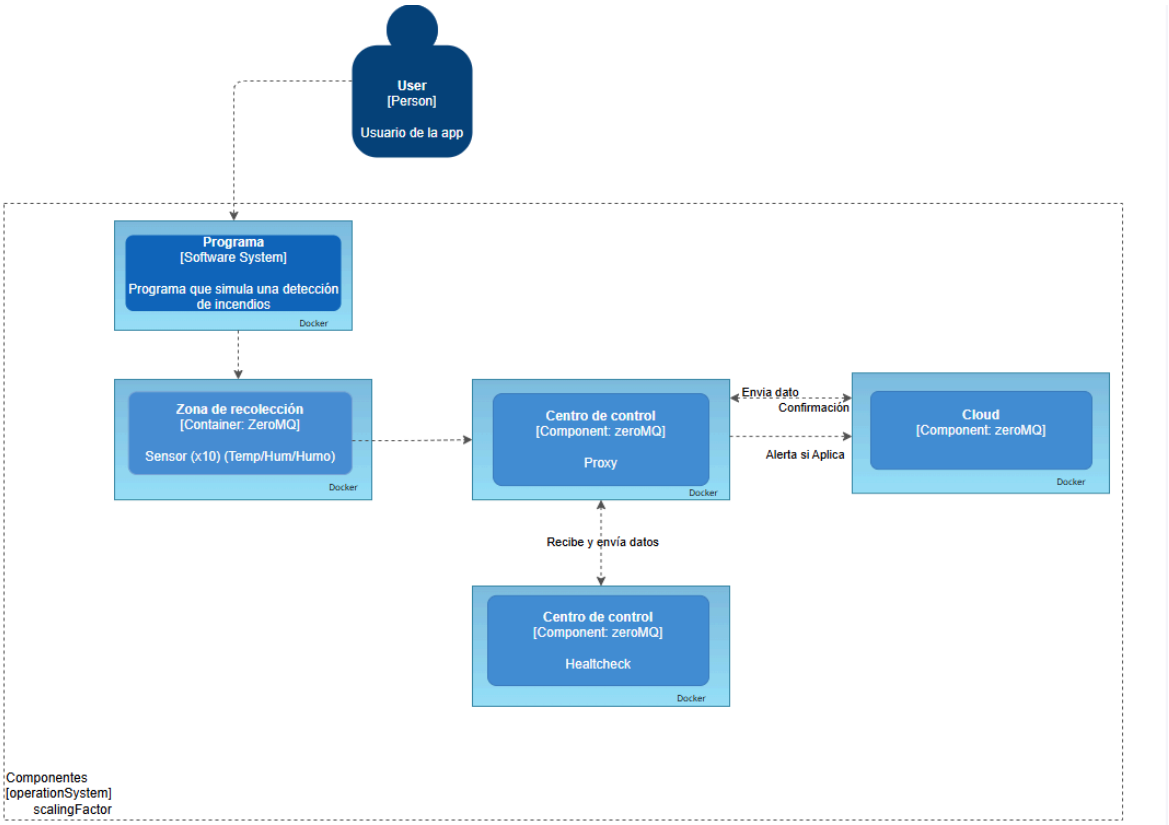


Imagen 8. Diagrama de Componentes C4

El diagrama de contenedores muestra la arquitectura del sistema dividiendo sus principales bloques de ejecución (contenedores). Este nivel detalla cómo los diferentes contenedores se comunican entre sí para cumplir el objetivo del programa.

Flujo y componentes:

El funcionamiento del sistema distribuido se organiza en una arquitectura basada en contenedores, donde cada componente ejecuta una función específica y se comunica a través de la librería ZeroMQ. A continuación, se describe el flujo completo de operación:

1. Inicio del sistema

   El programa principal inicia y levanta todos los contenedores necesarios mediante Docker. Estos incluyen:

   - Zona de Recolección (sensores y actuadores)
   - Centro de Control (proxy)
   - Health Check (monitor de fallas)
   - Nube (procesamiento y almacenamiento final)

2. Captura de datos en la Zona de Recolección

   Los sensores simulados de temperatura, humedad y humo comienzan a generar datos con una frecuencia determinada. Estos datos pueden ser normales, fuera del rango o erróneos, según una configuración de probabilidades.

   - Los datos generados se envían al Centro de Control usando el patrón PUSH de ZeroMQ.
   - Si un sensor de humo detecta presencia de humo:
     - Se activa automáticamente el aspersor (simulado).
     - Se genera una alerta que es enviada al Sistema de Calidad local.

3. Procesamiento en el Centro de Control

   El contenedor del Centro de Control actúa como un proxy que:

   - Recibe datos mediante el patrón PULL.
   - Valida los datos recibidos (descartando valores erróneos).
   - Calcula promedios de temperatura y humedad a partir de los sensores.
   - Si detecta valores fuera del rango, genera una alerta que se envía al Sistema de Calidad, sin embargo este sistema está implícito por lo que no está en un contenedor de docker aparte.
   - Posteriormente, transmite los datos procesados hacia la Nube usando el patrón Request-Reply de ZeroMQ.

4. Monitoreo de fallos con el Health Check

   Paralelamente, el Health Check Monitorea el estado del proxy en el Centro de Control mediante señales periódicas.

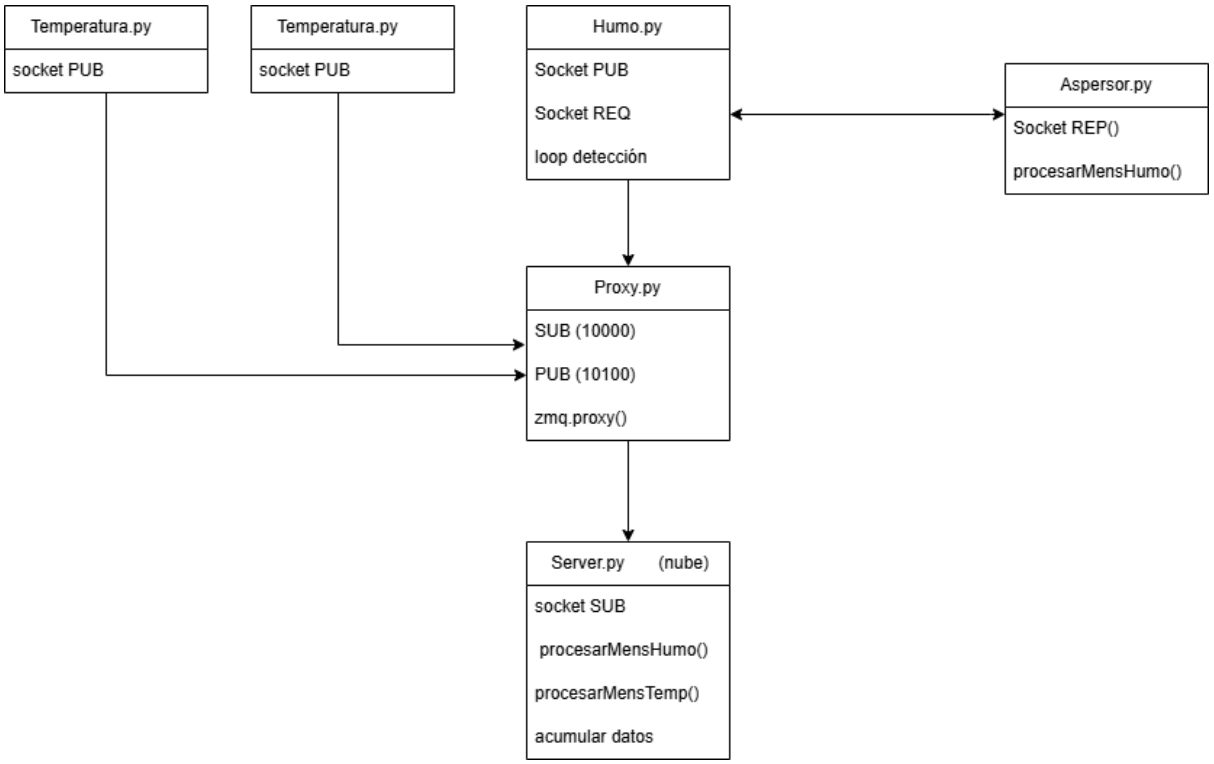
   - Si detecta que el proxy ha fallado, activa un proxy de respaldo alojado en otro contenedor, el cual toma el control automáticamente sin interrumpir el flujo del sistema.

5. Procesamiento final en la Nube

   La capa Cloud recibe los datos procesados y los almacena para análisis históricos.

   - Cada cierto tiempo, calcula la humedad mensual promedio a partir de los datos enviados desde el Centro de Control.
   - Si la humedad mensual es inferior al umbral establecido, se genera una alerta que es enviada al Sistema de Calidad de la Nube.

8. Diagrama de Código



Este diagrama representa el nivel más detallado de la arquitectura, mostrando las clases o componentes principales involucrados en el manejo de solicitudes, así como sus métodos asociados. En este caso, se detalla el componente de zona de recolección el cual son los 10 sensores y su interacción con la nube y el proxy.

Flujo y componentes:

1. Humo.py

- Rol: Simula sensor de humo
- Comunicación:
    - Envía datos al proxy usando ZeroMQ (PUB)
    - Envía alerta al Aspersor si detecta humo (REQ)

2. Temperatura.py / Humedad.py

- Rol: Simulan sensores de temperatura y humedad
- Comunicación:
    - Envían datos periódicos al proxy (PUB)

3. Aspersor.py

- Rol: Actúa como actuador en caso de alerta
- Escucha mensajes (REP) desde sensores de humo
- Acción: Muestra mensaje "ASPERSOR ENCENDIDO"

4. proxy.py

- Rol: Centro de control
- Usa zmq.proxy(frontend, backend)
    - SUB en puerto 10000
    - PUB en puerto 10100
    - Pasa los mensajes de sensores hacia el servidor (server)

5. server.py

- Rol: Nube
- Funciones:
    - procesarMensHumo(msg)
    - procesarMensTemp(msg)
    - Acumula, analiza y muestra resultados
- Suscriptor (SUB) a puerto del proxy (10100)

Conclusiones

- Arquitectura clara y distribuida:
  El sistema se dividió en tres capas (Recolección, Control y Nube) usando Docker, lo que facilita su despliegue y escalabilidad.

- Comunicación eficiente con ZeroMQ:
  Se usaron correctamente los patrones PUB-SUB y REQ-REP para enviar datos, alertas y activar el aspersor, garantizando un flujo de información confiable.

- Código modular y bien organizado:
  Los sensores, el proxy, el servidor y el aspersor están separados en paquetes específicos, lo que mejora la mantenibilidad y la claridad del sistema.

- Tolerancia a fallos implementada:
  Se integró un Health Checker que activa un proxy de respaldo si el principal falla, aportando resiliencia al sistema.

- Procesamiento y respuesta efectiva:
  El sistema detecta valores anómalos, genera alertas y activa el actuador de forma automática, cumpliendo el objetivo de monitoreo en tiempo real.

- ZeroMQ es una herramienta altamente eficiente para la comunicación entre procesos, diseñada con un enfoque minimalista pero poderoso. A lo largo del análisis, se destacan sus fortalezas en desempeño, escalabilidad y flexibilidad, pero también ciertas limitaciones en cuanto a fiabilidad, seguridad y presencia en el mercado laboral.

Lecciones Aprendidas

El estudio de los sistemas distribuidos permite entender que su diseño implica tanto retos técnicos como decisiones estratégicas. Algunas de las lecciones son:

1. La diversidad de herramientas permite construir sistemas robustos, pero cada componente debe elegirse según el contexto específico.

2. La complejidad conceptual es inevitable, con temas como sincronización, consenso y consistencia que exigen un sólido conocimiento teórico.

3. El desacoplamiento mediante arquitecturas orientadas a eventos mejora la escalabilidad y la flexibilidad del sistema.

4. La computación serverless simplifica la operación, pero introduce nuevos desafíos como la latencia de arranque y la pérdida de control sobre la infraestructura.

5. La observabilidad es esencial para mantener la salud del sistema, siendo necesario monitorear, registrar y rastrear eventos de forma centralizada.

6. El diseño de sistemas distribuidos implica compromisos, especialmente al aplicar el Teorema CAP, por lo que se debe priorizar según los objetivos del negocio.

Referencias y bibliografía

1. Definición de un sistema distribuido:

- Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms* (2nd ed.). Prentice Hall
- https://www.distributed-systems.net/index.php/books/distributed-systems/

2. Características de un sistema distribuido:

- Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2012). *Distributed Systems: Concepts and Design* (5th ed.). Pearson Education.
- https://www.distributed-systems.net/

3. Casos de Uso (Situaciones y/o problemas donde se pueden aplicar) de un sistema distribuido:

- Dean, J., & Ghemawat, S. (2008). *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM, 51(1), 107–113.

4. Casos de aplicación (Ejemplos y casos de éxito en la industria) de un sistema distribuido:

- Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). *The Google File System*. ACM SIGOPS Operating Systems Review, 37(5), 29–43.
- Dean, J., & Ghemawat, S. (2008). *MapReduce: Simplified Data Processing on Large Clusters*. Communications of the ACM, 51(1), 107–113.
- Cockcroft, A. (2015). *Migrating to Microservices*. Netflix Tech Blog.
- Meta Engineering Blog. (2020). *TAO: Facebook's distributed data store for the social graph*.
- Uber Engineering. (2019). *Scaling Uber's Real-Time Market Platform*.
- Spotify Engineering Blog. (2014). *Microservices at Spotify*.

5. Características zeroMQ

- Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. O'Reilly Media.
- The ZeroMQ Project. (2024). *ZeroMQ Documentation*. Disponible en: https://zeromq.org/
- Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. Prentice Hall.

6. Historia y evolución zeroMQ
- ZeroMQ Community. (2024). *ZeroMQ: History and Versions*. Disponible en: https://zeromq.org
- Tanenbaum, A. S., & Van Steen, M. (2007). *Distributed Systems: Principles and Paradigms*. Prentice Hall.GitHub – ZeroMQ Project: https://github.com/zeromq/libzmq

7. Casos de uso (Situaciones y/o problemas donde se pueden aplicar)
- Hintjens, P. (2013). *ZeroMQ: Messaging for Many Applications*. O'Reilly Media.
- The ZeroMQ Project. (2024). *Use Cases*. Disponible en: https://zeromq.or

8. Casos de uso (Situaciones y/o problemas donde se pueden aplicar)
- The ZeroMQ Project (2024). *Use Cases and Projects*. Disponible en: https://zeromq.org

- CERN Controls Software. (2017). *Using ZeroMQ in LHC Control Systems*. https://acc-co.github.io/

9. Que tan comun es el Stack
- https://www.star-history.com/#zeromq/libzmq

10. Matriz de análisis de Principios SOLID vs ZEROMQ
- Bass, Len; Clements, Paul; Kazman, Rick.
  Software Architecture in Practice (3rd ed.). Addison-Wesley, 2012.
  https://www.sei.cmu.edu
- ISO/IEC 25010 – Systems and software Quality Requirements and Evaluation (SQuaRE)
  Modelo estándar de atributos de calidad de software.
  ISO/IEC 25010
- ZeroMQ – The Guide
  Peter Hintjens (creador de ZeroMQ). Documento técnico que explica patrones de uso y capacidades.
  https://zguide.zeromq.org

11. Matriz de análisis de Tácticas vs ZeroMQ
- SEI – Carnegie Mellon: Architecture Tactics
  Artículos y presentaciones del Software Engineering Institute sobre tácticas.
  https://resources.sei.cmu.edu
- ZeroMQ Patterns & Reliability Models (ZeroMQ Guide & Community)
  Casos reales y documentación sobre patrones como PUSH/PULL, PUB/SUB, DEALER/ROUTER.
  https://zguide.zeromq.org/docs/chapter3/

LINKS DIAGRAMAS

1. Diagrama de Alto nivel

   https://app.diagrams.net/

2. Diagrama dinámico

   https://online.visual-paradigm.com/w/wwzfeagd/diagrams/#diagram:workspace=wwzfeagd&proj=0&id=8&type=ClassDiagram&width=11&height=8.5&unit=inch

3. Diagrama de Despliegue

   https://app.diagrams.net/

4. Diagrama de Paquetes UML

   https://app.diagrams.net/

5. Diagrama de Componentes

https://online.visual-paradigm.com/w/wwzfeagd/diagrams/#diagram:workspace=wwzfeagd&proj=0&id=9&type=ClassDiagram

6. Diagrama de Contenedores

   https://online.visual-paradigm.com/w/wwzfeagd/diagrams/#diagram:workspace=wwzfeagd&proj=0&id=10&type=ClassDiagram

7. Diagrama de Código:

   https://app.diagrams.net/