



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Refactorización de una infraestructura de bucles MAPE-K como microservicios

TRABAJO FIN DE GRADO

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Autor: Adriano Vega Llobell

Tutor: Joan Josep Fons Cors

Curso 2021-2022

Resum

???

Paraules clau: ????, ?????????, ????, ?????????????????

Resumen

???

Palabras clave: ?????, ???, ?????????????????

Abstract

???

Key words: ?????, ????? ?????, ?????????????????

Índice general

Índice general	V
Índice de figuras	VII
Índice de tablas	VII
<hr/>	
1 Introducción	1
1.1 Motivación	2
1.2 Objetivos	2
1.3 Estructura de la memoria	2
2 Contexto Tecnológico	3
2.1 Computación autónoma y bucles de control	3
2.1.1 Arquitectura para sistemas autónomos: Bucles MAPE-K	4
3 Arquitectura de la solución	9
3.1 Arquitecturas de <i>software</i>	9
3.1.1 Componentes de una arquitectura	9
3.1.2 Estilos arquitectónicos	11
3.2 Arquitectura de la solución	12
3.2.1 Distribución de los componentes	12
3.2.2 Conectando los servicios	13
3.2.3 Open API	22
3.2.4 Notificaciones	28
4 Caso de estudio: Sistema de climatización	29
4.1 Análisis	29
4.2 Diseño	29
4.2.1 Sonadas:	30
4.2.2 Propiedades de adaptación:	30
4.2.3 Monitores:	31
4.2.4 Reglas de adaptación	31
4.2.5 Efectores:	33
4.2.6 Configuración del sistema	33
4.3 Implementación	33
4.3.1 Telemetría	34
5 Conclusions	37
Bibliografía	39
<hr/>	
Apéndices	
A Configuració del sistema	41
A.1 Fase d'inicialització	41
A.2 Identificació de dispositius	41
B ??? ????????????? ???? ?	43

Índice de figuras

2.1	Un bucle de control genérico. Consta de cuatro actividades: Recopilar información, analizarla, decidir y actuar si procede.	3
2.2	Representación de un elemento autónomo. Distinguimos el recurso manejado y el manejador autónomo. El manejador es un bucle MAPE-K (<i>Monitor, Analysis, Planification, Execution y Knowledge</i>)	5
3.1	El servicio de monitorización representado como un componente. Ofrece una interfaz (<i>IMonitoringService</i>) y depende de otra para funcionar (<i>IKnowledgeService</i>).	10
3.2	Ejemplo de comunicación de dos componentes a través de un conector. . .	11
3.3	Arquitectura de un Bucle MAPE-K. Podemos apreciar el flujo de información y de control a lo largo de las etapas del bucle.	12
3.4	Diagrama con los componentes que forman nuestra arquitectura distribuida	13
3.5	Ejemplo del estilo arquitectónico C2 (<i>Components and Connectors</i>)	14
3.6	Representación de nuestra propuesta arquitectónica. Inspirado en Arquitectura Limpia (<i>Clean Architecture</i>). Las flechas negras representan las peticiones, y las moradas, las notificaciones.	15
3.7	Boceto inicial: queremos conectar el servicio de monitorización con la base de conocimiento para poder leer propiedades de adaptación.	16
3.8	Funcionamiento del sistema de objetos distribuidos	17
3.9	Representación de las colas de trabajo. Ejemplo de comunicación asíncrona dirigida.	20
3.10	Estrategia <i>publish/suscribe</i> : el <i>broker</i> actúa como intermediario en la comunicación <i>multicast</i>	20
3.11	Diseño del conector usando implementación Cliente - Servidor	27
4.1	Extracto de <i>logs</i> de una ejecución habitual.	34
4.2	Extracto de <i>logs</i> de una ejecución habitual.	35

Índice de tablas

3.1	Comparativa de los mecanismos de comunicación.	21
3.2	Especificación de la operación para obtener una propiedad del servicio de conocimiento.	22
3.3	Especificación de la operación para actualizar o crear una propiedad del servicio de conocimiento.	23
4.1	Sondas del sistema de climatización.	30

4.2	Propiedades de adaptación del sistema de climatización.	30
4.3	Monitores del bucle MAPE-K del sistema de climatización.	31
4.4	Reglas de adaptación del sistema de climatización.	33
4.5	Efectores del sistema de climatización.	33

CAPÍTULO 1

Introducción

Debido al avance de las tecnologías en las últimas décadas, y a la penetración del *software* en todos los ámbitos de nuestras vidas, cada vez tenemos sistemas más complejos y con requisitos de disponibilidad más altos. Por ejemplo, en caso de tener una tienda online, necesitamos asegurar que la tienda está disponible el mayor tiempo posible. Cuanto más tiempo pase "caída", menos potenciales clientes nos comprarán, y perderemos ingresos.

Por otro lado, queremos también que nuestro sistema sea capaz de adaptarse a picos de demanda, aumentando su capacidad de cómputo cuando tengamos mayor afluencia de clientes. Por ejemplo, en temporadas de rebajas como *black friday*. Operar sistemas capaces de escalar, deriva en sistemas complejos. Como no es viable tener a operarios pendientes del estado del sistema para llevar a cabo estas adaptaciones. Deben hacerse automáticamente.

En el ámbito de la computación autónoma encontramos el concepto de sistemas **autoadaptativos**: aquellos capaces de ajustar su propio comportamiento en base a cambios en su entorno de operación. Se caracteriza por dotarlos con capacidades para razonar sobre su estado de operación y su entorno. En base a estos parámetros, el sistema puede intuir que debe reconfigurarse para cumplir con los objetivos que tiene marcados. Para ello, en base a una serie de estrategias predefinidas, es capaz de elegir su siguiente configuración. [1]. Esto conlleva mover a tiempo de ejecución las decisiones de arquitectura y funcionalidad. Con ello, buscamos permitir un comportamiento dinámico del sistema. [2].

En este trabajo se quiere abordar la división de un servicio monolítico y adaptarlo para su funcionamiento en entornos en la nube. Para ello, se quiere extraer su funcionalidad en distintos microservicios. Es decir, se quiere **cambiar la topología** de la solución. Se trata de un cambio importante en la arquitectura de la solución.

En concreto, se trata de un servicio que implementa un bucle de control MAPE-K [3, 4], una para la implementación de sistemas autónomos propuesta inicialmente por IBM. El bucle se encarga de gestionar un **recurso manejado** en base a unas **políticas** definidas por el administrador del sistema. Las políticas

La idea es separar cada una de sus etapas en microservicios individuales. De esta forma, podemos desarrollarlas de forma independiente entre ellas, replicarlas para mejorar su escalabilidad, o sustituirlas por implementaciones distintas, etc.

Para desarrollar el trabajo, propusimos el siguiente plan:

- Cada etapa del bucle será un microservicio distinto. Extraeremos cuatro microservicios distintos: Planificador, Analizador,

Por tanto, los conectores elegidos para comunicar los microservicios han sido más centrados en comunicar con las APIs públicas que expone cada uno.

1.1 Motivación

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

1.2 Objetivos

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

1.3 Estructura de la memoria

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

CAPÍTULO 2

Contexto Tecnológico

2.1 Computación autónoma y bucles de control

Según [3], la **computación autónoma** tiene como objetivo dotar a los sistemas de **autonomía** en su operación; capacidades para gestionarse a si mismos. Es decir, deberán adaptarse a los distintos escenarios que puedan darse durante su ejecución. Con esto, buscamos alcanzar una reducción en el coste de operación y hacer más gestionable la complejidad de los sistemas.

Estas adaptaciones se realizan en base a directivas de alto nivel proporcionadas por un humano: el humano fija los objetivos que el sistema debe alcanzar; y este deberá adaptarse para lograrlo, si es posible. Siguiendo con el ejemplo de la página web, el operador humano podría definir un máximo de carga por cada instancia. Entonces, cuando se supera el umbral, el sistema podría decidir que se requiere una acción correctiva que consista en desplegar nuevas instancias del servicio cuando haya muchos accesos concurrentes. Cuando la carga de los servicios baje, podemos eliminarlas.

Para implementar estas capacidades de adaptación, recurriremos a la teoría de control y el **bucle de control** (o *feedback loop*). [2] Se trata de un proceso iterativo compuesto por cuatro actividades (figura 2.1):

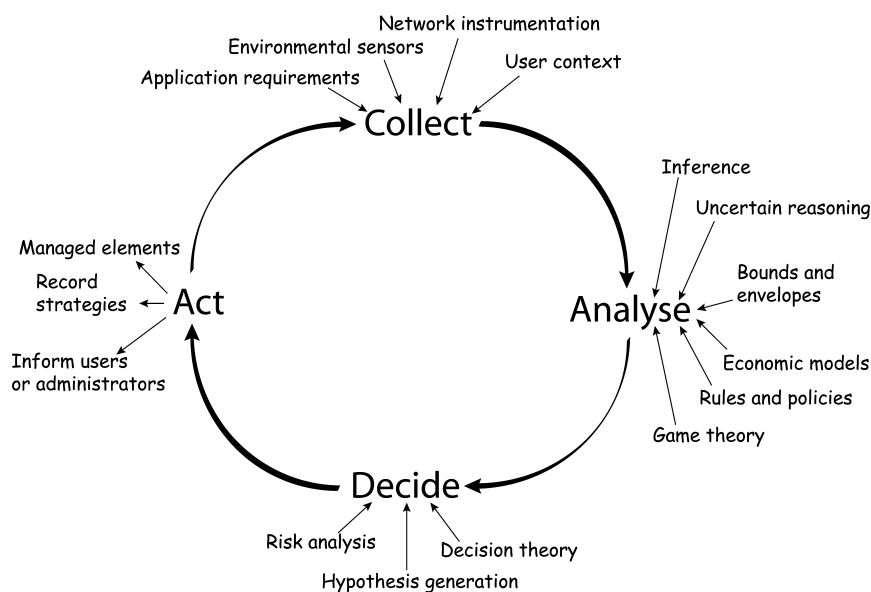


Figura 2.1: Un bucle de control genérico. Consta de cuatro actividades: Recopilar información, analizarla, decidir y actuar si procede. Obtenida de [5].

- **Recopilar información:** El bucle **monitoriza** el estado del sistema a través de **sondas**. Estas reportan información del sistema y del entorno de ejecución. Pueden ser métricas del rendimiento del sistema, estado de los componentes, etc.
Estos datos en bruto deben ser limpiados, filtrados y agregados. Si se considera que son relevantes, se almacenan para informar las siguientes etapas del bucle.
- **Analizar:** A partir de las propiedades de adaptación, la etapa de análisis debe identificar **síntomas**: indicadores de una situación que requiera de nuestra atención. Puede ser mediante heurísticas que hayamos configurado, análisis estadístico **y cosas así**. Un ejemplo de síntoma sería "uso de CPU elevado", "número elevado de mensajes encolados en un sistema de mensajería" **entre otras**.
- **Decidir:** A partir de los síntomas, el bucle debe determinar si es necesario tomar alguna acción correctiva. **Planifica** las acciones que se llevarán a cabo para que el sistema se adapte y alcance una configuración deseable. Por ejemplo, si hay muchos mensajes encolados, se podría solicitar el iniciar otra instancia del servicio que los consuma y procese en paralelo.
- **Actuar:** Si el bucle ha planificado alguna acción, se intentará **ejecutar** en esta etapa final. Mediante **efectores** en el sistema, el bucle es capaz de cambiar la configuración actual del mismo. Dependiendo del éxito de esta etapa, la adaptación se lleva a cabo o no. Finalizada esta, se vuelve a recopilar información y el bucle continúa iterando.

Este tipo de proceso está presente en gran variedad de contextos como puede ser operación de plantas industriales, en procesos naturales, etc. **Citar TFM planta embalaje**. En la ingeniería de *software*, encontramos diversas aplicaciones de los bucles de control. Pero normalmente están implícitos en la implementación. [2]

los bucles de control pueden ser implícitos, dentro del código y las condiciones, o explícitos. [2] Lo ideal es contar con bucles externos, esto nos permite separar la funcionalidad de las capacidades de adaptación. Esto facilita la implementación.

Garlan et al. also advocate to make self-adaptation external, as opposed to internal or hard-wired, to separate the concerns of system functionality from the concerns of self-adaptation [9,16].

Hablar de agentes autónomos como aplicación práctica. [6]

2.1.1. Arquitectura para sistemas autónomos: Bucles MAPE-K

Un estilo arquitectónico muy representativo de este tipo de sistemas es el basado en bucles MAPE-K [3, 4] propuesto por IBM. Se trata de una arquitectura para sistemas distribuidos autónomos que requieran del mínimo de intervención humana para operar. Nace con el objetivo reducir en el coste de operación y hacer más gestionable la complejidad de los sistemas.

Estos sistemas son capaces de auto-gestionarse en base a **políticas**. Las políticas son un conjunto de objetivos de alto nivel que definen los usuarios encargados del sistema. El sistema debe tratar de alcanzarlos durante su funcionamiento. Además, estos motivan los cambios en el sistema, que trata de adaptarse para alcanzarlos.

Sus componentes principales son los **elementos autónomos**. Cada uno de estos es capaz de autogestionarse, y colaborar en conjunto con el resto de elementos autónomos del sistema para alcanzar los objetivos. **¿Agent based?** A su vez, estos pueden dividirse en dos partes: los recursos manejados y un manejador autónomo (el bucle de control).

Los **recursos manejados** son las unidades de funcionalidad del sistema. Puede ser cualquier tipo de recurso, *hardware* o *software*. Para dotarlas de capacidad de autoadaptación, las emparejamos con un **manejador autónomo**, el bucle de control. Como es un componente distinto al que implementa la funcionalidad, es entonces de tipo externo. Gestiona al recurso en base a la información que recoge del entorno de ejecución y las políticas que guían su adaptación.

Para poder ser gestionado externamente, el recurso debe implementar *touchpoints* (*¿puntos de contacto?*): interfaces que permiten al bucle de control obtener información del estado del sistema y cambiar su configuración. Existen dos tipos de *touchpoints*: sondas y efectores.

En la figura 2.2 tenemos una representación de un elemento autónomo. Distinguimos las dos partes: el manejador y el recurso. El manejador está acoplado al recurso a través de sus sensores y efectores. Podemos apreciar que **siete** componentes distintos conforman el bucle: [3]

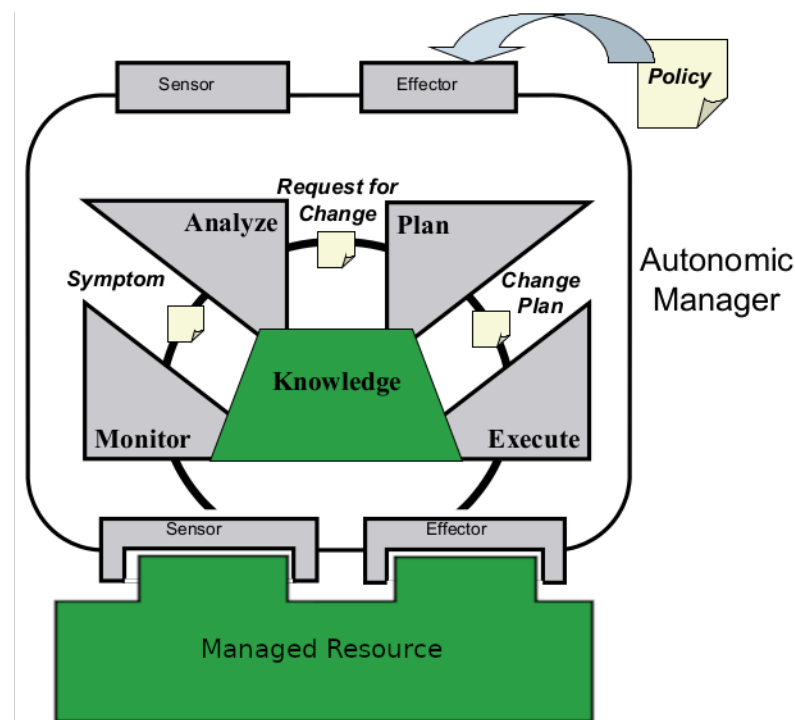


Figura 2.2: Representación de un elemento autónomo. Distinguimos el recurso manejado y el manejador autónomo. El manejador es un bucle MAPE-K (*Monitor, Analysis, Planification, Execution y Knowledge*). Basada en imagen de [3].

Sondas

Para poder monitorizar el recurso y su entorno deberemos **instrumentarlos**. Consiste en implementar **sondas** que expongan datos relevantes a los monitores del bucle. Pueden capturar y transmitir cualquier métrica que queramos controlar: *health checks*, rendimiento del servicio u otras propiedades del sistema.

Monitor

El monitor recibe las mediciones de las sondas. Se encarga de recogerlas, agregarlas y filtrarlas para determinar si ha ocurrido un evento relevante que deba ser reportado. Si se

considera que lo son, estos valores se almacenan como propiedades de adaptación en la base de conocimiento. [7] Por ejemplo, en un sistema de climatización, si la temperatura de una habitación cambia.

Base de conocimiento

La base de conocimiento (*knowledge base*) está compuesta por una o más fuentes de información que el bucle tiene a su disposición. En ellas, se almacenan las **propiedades de adaptación**. En conjunto, estas propiedades conforman un modelo abstracto del sistema, que describe su estado pasado y actual: componentes, conexiones entre ellos y su configuración. [1]

El bucle del control trabaja con un modelo del sistema de alto nivel [1]. Esto le permite definir las adaptaciones desacoplándose de los elementos. Los sistemas adaptativos se basan principalmente en bucles de control. trabajan sobre system models - and in particular, architectural models - are maintained at run time and used as a basis for system reconfiguration and repair [1]

Es una arquitectura knowledge-driven [8].

El conocimiento informa a todas las etapas del bucle de control. Por tanto, se trata de un componente transversal.

Analizador

En base a las propiedades de adaptación, podemos razonar sobre el estado actual del sistema y detectar situaciones que requieran de una acción correctiva. Para ello contamos con el módulo de análisis.

Una aproximación para implementarlo es mediante **reglas de adaptación**, compuestas por una condición y una acción. Las reglas se suscriben a cambios de las propiedades de adaptación. Cuando ocurra alguno, se evalúa su condición. Si esta se cumple, se ejecuta la acción asociada. En caso contrario, no hará nada.

Siguiendo con el ejemplo del sistema de climatización, un ejemplo de regla sería...

La acción de la regla describe una **propuesta de cambio** en la configuración del sistema. Estas se formulan en base a **operadores arquitectónicos**. [1] Dependiendo del estilo arquitectónico de nuestro sistema, tendremos disponibles una serie de operaciones para alterar su arquitectura.

Por ejemplo, nuestro recurso manejado podría estar implementado como microservicios. En este caso, los operadores podrían consistir en desplegar o eliminar servicios, establecer conexiones entre los servicios, eliminarlas, o cambiar las propiedades de configuración del servicio. [4]

Planificador

Si alguna regla se dispara, el planificador recibe los cambios propuestos. Comparando sus acciones con el modelo que tenemos del estado del sistema (las propiedades de adaptación), determina si todavía es necesario ejecutar estas acciones. También comprobará si es seguro aplicarlas, ya que no deben dejar el sistema en un estado inconsistente. En caso de que las propuestas sean válidas, estas se agruparán en un **plan de adaptación**.

Ejecutor

En la etapa final del bucle tenemos al ejecutor. El ejecutor interactúa con los efectores del sistema manejado para llevar a cabo las acciones planificadas. Para ello, traduce las acciones de alto nivel (nivel de arquitectura) a acciones de más bajo nivel (en términos del propio sistema). [1]

Efectores

Los **efectores** sirven para modificar el estado del sistema manejado. Pueden ser ficheros de configuración, comandos, *endpoints*, etc.

Además, podemos observar que el propio manejador expone sensores y efectores, lo que permite que sean controlados por **manejadores autónomos orquestadores**. Estos gestionan a un nivel superior uno o más elementos autonómicos. Son por tanto, elementos componibles.

¿Hablar del nivel en el que se encuentra el bucle de control? Sistema, infraestructura, mixto, mesh [9]

Hablar del *human manager*, la capa superior al sistema. Emite las políticas y monitoriza su funcionamiento a través de las sondas del bucle orquestador.

Hablar de human in the loop: solicitamos la intervención del humano cuando no contamos con suficiente información para tomar una acción correctiva.

CAPÍTULO 3

Arquitectura de la solución

En este capítulo vamos a describir la arquitectura que hemos diseñado para distribuir el bucle MAPE-K. Partimos de un sistema con una **división funcional ya definida**, por lo que será sencillo delimitar los componentes. El foco de este capítulo serán entonces los **conectores de software**. Necesitamos establecer qué estrategias de comunicación utilizaremos para comunicar los componentes.

Comenzaremos dando una breve introducción a las arquitecturas de *software* y los elementos que las componen. Después, describiremos la arquitectura de nuestra solución y el proceso que hemos seguido para llegar hasta ella.

3.1 Arquitecturas de *software*

Según [8], la **arquitectura de un sistema *software*** es el conjunto de todas las **decisiones principales de diseño** que se toman durante su ciclo de vida; aquellas que sientan las bases del sistema. Estas afectan a todos sus apartados: la funcionalidad que debe ofrecer, la tecnología para su implementación, cómo se desplegará, etc. En conjunto, definen una pauta que guía (y a la vez refleja) el diseño, la implementación, la operación y la evolución del sistema.

Todos los sistemas *software* cuentan con una. La diferencia radica en si esta ha sido diseñada y descrita explícitamente o ha quedado implícita en su implementación. [8] En el segundo caso es probable que, con el paso del tiempo, se “erosione” su arquitectura: se implementan funcionalidades sin respetar la estructura. También se olvida el por qué de ciertas decisiones. En general, se vuelve más difícil de mantener. Se convierte en una “gran bola de barro”. [10]

Por tanto, es vital dedicar tiempo para definirla atendiendo a las necesidades de nuestro sistema. Una buena arquitectura es capaz de dotar de estructura a nuestro sistema. [?] Mientras se respeta la arquitectura, y se mantenga actualizada, esta estructura. Una buena arquitectura nos ofrece una serie de ventajas, como facilitar su desarrollo, mayor extensibilidad.

3.1.1. Componentes de una arquitectura

Otra posible definición de arquitectura la encontramos en el estándar IEEE 42010-2011 [11]: es “*un conjunto de conceptos o propiedades fundamentales, personificados por sus elementos, sus relaciones, y los principios que guían su diseño y evolución*”.

Podemos describirlas entonces usando tres conceptos: [12]

- **Componentes** (o elementos): Son las piezas fundamentales que conforman el sistema. Representan las unidades de funcionalidad de la aplicación. Se utilizan para describir *qué* partes componen el sistema. Por ejemplo: un módulo, un servicio web...
- **Forma**: El conjunto de propiedades y relaciones de un elemento con otros o con el entorno de operación. Describe *cómo* está organizado el sistema. Por ejemplo: un servicio A contacta con otro, B, usando una llamada HTTP.
- **Justificación**: Razonamiento o motivación de las decisiones que se han tomado. Responden al *por qué* algo se hace de una manera determinada. Normalmente no pueden deducirse a partir de los elementos y la forma, por lo que es necesario describirlos explícitamente.

Elementos

El primer tipo de elemento que debemos tratar son los componentes. Según [8], los **componentes** son “elementos arquitectónicos que encapsulan un subconjunto de la funcionalidad y/o de los datos del sistema”. Dependiendo de las características de nuestro sistema (y del nivel de abstracción que usemos) pueden tomar distintas formas: objetos, módulos dentro un mismo proceso, servicios distribuidos, etc.

Los componentes exponen una **interfaz** que permite acceder a la funcionalidad o datos que encapsulan. A su vez, también declaran una serie de **dependencias** con interfaces de otros. Allí se incluyen todos los elementos que requieren para poder funcionar. En la figura 3.1 tenemos un ejemplo. *Monitoring Service* expone la interfaz *IMonitoringService*. Para poder funcionar, depende de un componente que ofrezca *IKnowledgeService*.

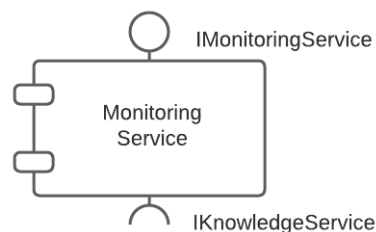


Figura 3.1: El servicio de monitorización representado como un componente. Ofrece una interfaz (*IMonitoringService*) y depende de otra para funcionar (*IKnowledgeService*).

Por si solos, estos componentes independientes no aportan mucho valor. Más bien son la unidad básica de composición: podemos combinar varios de ellos para que trabajen conjuntamente y realicen tareas más complejas. Así, podemos **componer sistemas**. [13] La integración y la interacción entre ellos son aspectos clave que debemos abordar.

Para que dos o más componentes puedan interactuar, necesitamos definir un mecanismo de comunicación. Recurrimos entonces a los **conectores**. Se trata de elementos arquitectónicos que nos ayudan a definir y razonar sobre la comunicación entre componentes. En la figura 3.2 mostramos una representación de la necesidad de comunicación entre dos componentes a través de un conector. No se ha especificado todavía ningún detalle sobre cómo se implementará. Así, podemos estudiar la arquitectura y elegir los mecanismos adecuados para cada interacción del sistema. [8].

Internamente, los conectores están compuestos por uno o más **conductos** o canales. A través de estos se lleva a cabo la comunicación entre los componentes. Hay una gran variedad de conductos posibles: comunicación interproceso, a través de la red, etc. Clasificamos los conectores según la complejidad de los canales que utilizan [13]:

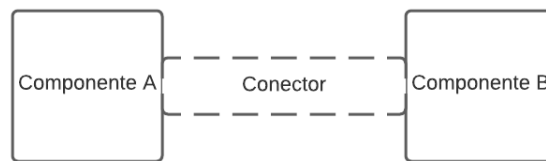


Figura 3.2: Ejemplo de comunicación de dos componentes a través de un conector.

- **Conectores simples:** solo cuentan con un conducto, sin lógica asociada. Son conectores sencillos. Suelen estar ya implementados en los lenguajes de programación. Por ejemplo: una llamada a función en un programa o el sistema de entrada / salida de ficheros.
- **Conectores complejos:** cuentan con uno o más conductos. Se definen por composición a partir de múltiples conectores simples. Además, pueden contar con funcionalidad para manejar el flujo de datos y/o control. Suelen utilizarse importando *frameworks* o librerías. Por ejemplo: un balanceador de carga que redirige peticiones a los nodos.

Por tanto, cuando hayamos decidido que dos componentes necesitan comunicarse, es momento de evaluar qué mecanismo de comunicación es más adecuado. Basándonos en nuestros requisitos, la arquitectura ya definida, y los mecanismos de despliegue que queremos usar, elegimos el conector apropiado. Podemos orientarnos con taxonomías como la de [13].

Forma

TODO: - ¿Borrar? Innecesario

Justificación

TODO: - ¿Borrar? Innecesario

Una vez definidos los componentes, los conectores y las relaciones entre ellos, tendremos una representación del sistema. Pero se trata de una imagen incompleta. No cuenta con ciertos detalles del contexto que nos ayudan a entenderlo mejor. Un ejemplo podría ser qué alternativas se consideraron y por qué se descartaron en favor de la elegida. Tampoco contamos con detalles minuciosos que puedan guiar mejor la implementación.

Es decir, requerimos de un concepto adicional para describirlos en nuestra arquitectura: se trata de la **justificación**. [12] Nos aporta detalles más precisos sobre el sistema que no se pueden representar como elementos o forma.

3.1.2. Estilos arquitectónicos

TODO: - ¿Borrar? Innecesario

Podemos agrupar decisiones principales.

3.2 Arquitectura de la solución

Como comentamos en el capítulo 1, el objetivo del trabajo es transformar un servicio monolítico en un sistema distribuido basado en microservicios. Se trata de un cambio arquitectónico importante. Queremos por tanto diseñar una estrategia ingenieril para llevar a cabo la migración; teniendo en cuenta las particularidades del sistema.

El servicio en cuestión implementa un **bucle de control MAPE-K**[3, 4], que ya describimos en la sección 2.1.1.

En esta sección presentaremos nuestra propuesta arquitectónica para adaptar el bucle para entornos en la nube.

Buscar libros de descomposición de monolitos en microservicios.

3.2.1. Distribución de los componentes

Actualmente, el bucle está muy acoplado a los modelos de sus recursos manejados. Todo corre bajo el mismo proceso: el bucle, los monitores, sus reglas de adaptación y demás elementos específicos de la solución... Ese proceso solo podrá manejar aquellos sistemas cuyos módulos tenga cargados. En la figura 3.3 presentamos otra vista de la arquitectura del bucle.

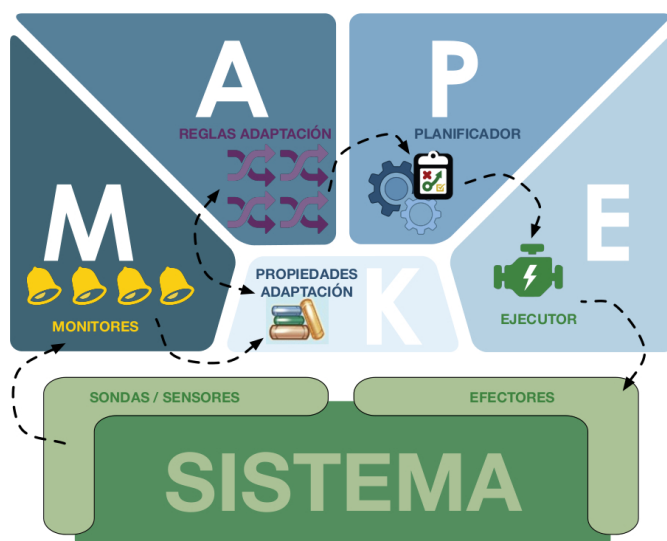


Figura 3.3: Arquitectura de un Bucle MAPE-K. Podemos apreciar el flujo de información y de control a lo largo de las etapas del bucle. Obtenida de [7]

Partimos entonces el objetivo de desacoplarlo. Así, podremos desplegarlo y usarlo de forma agnóstica al recurso manejado. La misma infraestructura podrá aprovecharse para manejar varios sistemas simultáneamente (*multi-tenancy*). La idea es implementarlo a nivel de sistema[9], por lo que se desplegará al mismo con los microservicios del recurso manejado.

Como veremos a continuación, cada uno de sus componentes es candidato a convertirse en un microservicio individual.

Por la descripción de ambos componentes, vemos que existe una clara división de dominios y responsabilidades. Esto nos ayuda a determinar que ambos componentes pueden desplegarse por separado. **REFERENCIA 'Building Microservices' Sam Newman**

Por suerte, partimos de un sistema existente, con una arquitectura bien definida y documentada. Conocíamos el rol de cada uno de los componentes del servicio y sus requisitos. Así que, el primer problema al que nos enfrentamos estaba relacionado con la distribución de los servicios. ¿Cómo definimos las fronteras entre cada uno de ellos? ¿Qué componentes debe abarcar cada microservicio?

La primera decisión que tomamos fue desacoplar el bucle de los sistemas. Buscábamos desarrollar microservicios agnósticos a la solución manejada. Por ello, vamos a identificar distintos **niveles de componentes** *Imagen que separa el bucle de la lógica de la solución*. Esto nos permitiría dar servicio a varios sistemas distintos con la misma infraestructura. Multi-tenancy.

Otra decisión que tomamos fue separar cada etapa del bucle en su propio servicio. Así podríamos independizarlas y escalarlas individualmente.

Una vez determinadas las fronteras entre los microservicios, hemos definido los componentes de nuestro sistema.

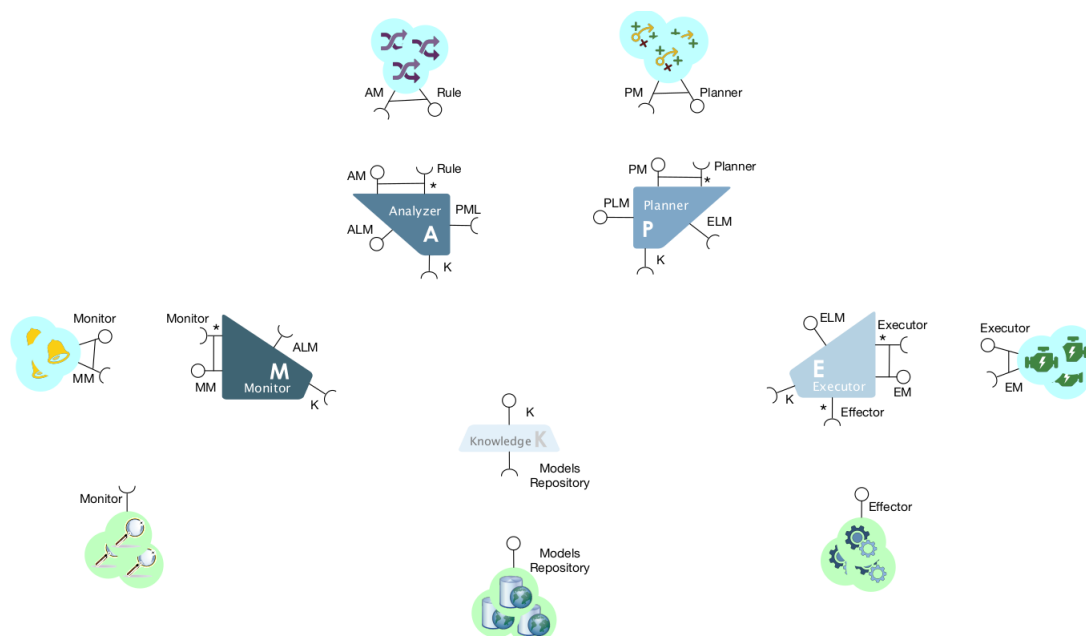


Figura 3.4: Diagrama con los componentes que forman nuestra arquitectura distribuida

Figura 3.4: Agrupar los servicios para poder aumentar zoom y hacerlo más legible. Añadir línea de división entre la capa del bucle y el dominio del recurso manejado.

3.2.2. Conectando los servicios

El siguiente problema al que nos enfrentamos está relacionado con la comunicación: si dividimos estos componentes en microservicios, ¿cómo hacemos para que se comuniquen? Hay que tener en cuenta que estos pueden estar desplegados y replicados en distintas máquinas. No podemos asumir que están en el mismo *host*.

Aprovechando la separación entre bucle de control y el dominio del recurso, investigamos posibles arquitecturas. Nos decantamos por **arquitecturas de servicios jerarquizados**. Queríamos explotar esta separación para mantener al bucle aislado del dominio de la solución. Dimos con el estilo arquitectónico C2 (*components and connectors*)[14, 15], en el que nos hemos inspirado.

Jerarquías de microservicios: Arquitectura C2 y arquitectura limpia

Este estilo organiza sus componentes en jerarquías o capas: cada servicio se encuentra en un nivel determinado, según su nivel de abstracción. En las capas inferiores, se encuentran los servicios más externos, más "acoplados" al entorno. Por ejemplo, aquellos servicios que requieran de acceder al sistema de ficheros, estarían en esta capa. Por otro lado, en las capas superiores se encuentran los servicios en niveles de abstracción superior, que dependen lo mínimo del entorno.

En cuanto a la comunicación, un componente solo debe contactar con sus vecinos inmediatos (en una capa superior o inferior). Esto evita que el servicio pueda contactar con otras capas, limitando su alcance y su conocimiento del despliegue del sistema. Además, dentro del mismo nivel no pueden contactar entre ellos. Según la dirección de la comunicación, se emplean mecanismos distintos (figura 3.5):

- **Peticiones** (*requests*): Se trata de solicitudes a un servicio para que ejecute una acción. Un componente se comunica directamente con un vecino en una capa superior. La petición viaja de "fuera hacia dentro" en cuanto al nivel de abstracción. Por ejemplo, una petición de un cliente a un servicio web podría estar bajo esta categoría.
- **Notificaciones**: Representan eventos ocurridos en el sistema. Un componente de más arriba en la jerarquía (más interno) envía un mensaje hacia abajo, sin especificar receptor. Todos los servicios por debajo lo recibirán, y decidirán si tratarlo o no. Esto evita que nuestro servicio se acople a aquellos que están por debajo (son más concretos). Un ejemplo sería notificar al resto de servicios sobre la creación de un nuevo usuario.

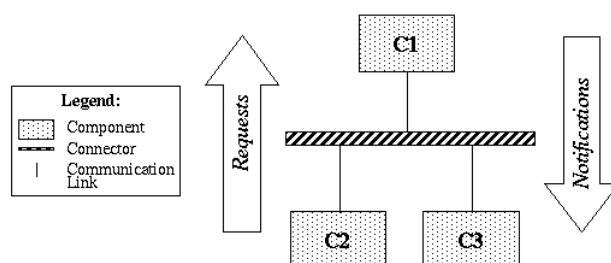


Figura 3.5: Ejemplo del estilo arquitectónico C2 (*Components and Connectors*). [15]

Basándonos en este estilo, definimos las capas de nuestro sistema. Esto nos permitió dividir los microservicios en niveles y elegir los conectores más adecuados para cada tipo de comunicación.

Distinguimos cuatro niveles distintos, de menor nivel de abstracción a mayor:

- **Nivel del recurso manejado**: En este nivel se encuentran las sondas y efectores. Son los elementos que tienen más contacto con el recurso manejado. Hacen de intermediarios entre este y el resto del bucle, para reducir su acoplamiento.
- **Nivel de específico solución**: En esta capa se encuentran componentes del bucle específicos para el dominio del recurso manejado. Monitores específicos, reglas de adaptación... No los incluimos en el mismo nivel que las sondas y efectores porque necesitamos comunicar con ellos. Además que guardan más relación con el bucle que con el recurso manejado.

- **Nivel del bucle:** Aquí se encuentran los servicios de las etapas del bucle: servicio de monitorización, análisis, planificación y ejecución. Esta capa debe ser agnóstica al dominio de los recursos manejados. Además, actúa como intermediario entre los servicios de la solución y el conocimiento. Limitan cómo acceder a él.
- **Conocimiento:** Es la capa más interna y la base de la arquitectura. No depende de ningún otro componente, por lo que tiene el nivel de abstracción más alto. Todos los componentes del nivel del bucle dependen de ella para funcionar.

Habiendo definido esta jerarquía, vimos ciertas similitudes con arquitecturas *domain driven*, como *Clean Architecture*. [16] En ella, el sistema se organiza en base a una **regla de dependencia**: “la dependencia entre los componentes solo puede apuntar hacia dentro, hacia políticas de alto nivel”. Es decir, la arquitectura se organiza en capas concéntricas. En el centro se encuentra el dominio, con el mayor nivel de abstracción. Este no tiene dependencias con ninguna capa exterior. Por otro lado, cada capa más externa tiene dependencias sólo con la capa a la que envuelve. Sólo puede comunicarse con componentes dentro de esta.

Basándonos en la descripción anterior, nuestra capa central será la del conocimiento. A partir de ahí, cada nivel superior dependería de aquel al que “envuelve”: el bucle al conocimiento, la solución al bucle... Por tanto, para que nuestra arquitectura sea más comprensible, optamos por representarla los diagramas de *Clean Architecture* para representarlo. En la figura 3.6 mostramos el resultado:

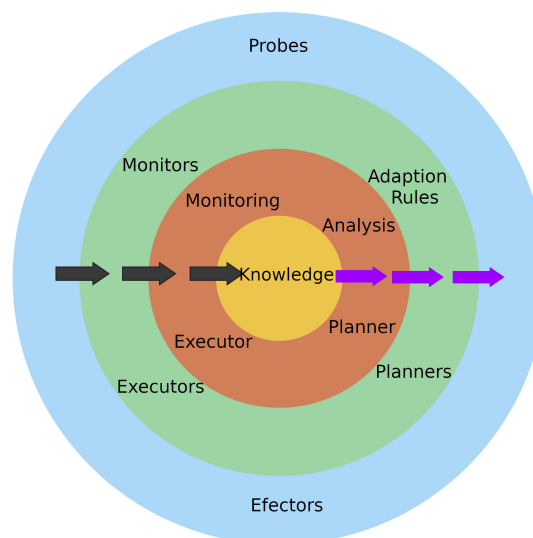


Figura 3.6: Representación de nuestra propuesta arquitectónica. Inspirado en Arquitectura Limpia (*Clean Architecture*). Las flechas negras representan las peticiones, y las moradas, las notificaciones.¹

Definiendo los mecanismos de comunicación

Como comentamos antes, vamos a inspirarnos en los mecanismos de comunicación descritos por C2: las peticiones y notificaciones. Pero estos no cubren todas nuestras necesidades. Hay dos casos que no están contemplados: la comunicación del módulo de análisis con el planificador, y la del planificador con el ejecutor. Ambos módulos se en-

¹Imagen original de arquitectura limpia obtenida de: <https://threedots.tech/post/ddd-cqrs-clean-architecture-combined/>

cuentran en la misma capa. Y, como dependen del conocimiento, no podemos moverlos a una superior para utilizar notificaciones.

Las notificaciones no nos sirven, ya que la comunicación es entre dos módulos específicos. Aunque nos interesa el desacoplamiento entre módulos que ofrecen. Las peticiones tampoco casan del todo, ya que requerimos desacoplar los módulos. Deberían mantener su independencia en el mayor grado posible. Por ello, requerimos de un tercer patrón de comunicación. Una combinación de ambos: las peticiones asíncronas.

Los tres patrones de comunicaciones que usaremos entonces son:

- **Peticiones síncronas:** Comunicaciones síncronas dirigidas a un servicio determinado. Solo permitidas entre servicios de una capa más externa a un servicio en la capa interior adyacente.
- **Peticiones asíncronas:** Comunicaciones asíncronas dirigidas a un tipo de servicio determinado. Se trata de peticiones de trabajo asíncronas: se envían y el destinatario lo procesará cuando pueda. El cliente continuará su ejecución, sin esperar respuesta. *fire and forget*.

Para evitar el acoplamiento entre los componentes, deberemos buscar un conector que permita enviar el mensaje sin conocer específicamente al destinatario.

Este mecanismo de comunicación solo está permitido entre elementos del mismo nivel.

- **Notificaciones:** Comunicaciones asíncronas no dirigidas. El servicio publica un evento que potencialmente recibirán todos los servicios en la capa exterior adyacente. El cliente lo envía y continua su ejecución, sin esperar respuesta.

Conectores

Una vez determinadas las necesidades de comunicación de nuestro sistema, debemos buscar los conectores adecuados. Seguimos la estrategia descrita en [8] para elegir conectores; y nos basamos en los patrones de comunicación en sistemas distribuidos descritos en [17].

Comenzamos investigando las peticiones síncronas. Tomemos por ejemplo la comunicación entre el servicio de monitorización (*monitoring service*) y el servicio de conocimiento (*knowledge service*). Recordemos que el servicio de conocimiento almacena todas las propiedades de adaptación. El resto de servicios necesitan consultarlas y actualizarlas durante su funcionamiento. En la figura 3.7 representamos inicialmente ambos componentes y un conector, sin especificar de qué tipo será.

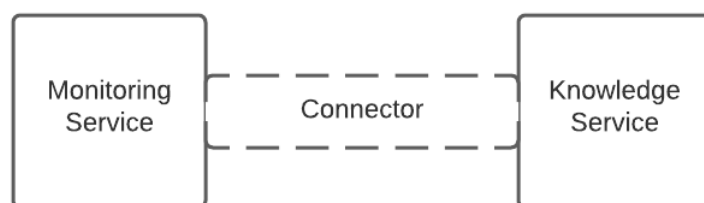


Figura 3.7: Boceto inicial: queremos conectar el servicio de monitorización con la base de conocimiento para poder leer propiedades de adaptación.

El siguiente paso es identificar qué interacciones debe existir entre ambos componentes. En este caso, el servicio de monitorización debe contactar con el servicio de conocimiento para leer y actualizar el valor de las propiedades. Por tanto, existen operaciones de lectura y escritura de los datos. Además, como es una comunicación "descendente" (*monitoring service* está en la capa superior), el patrón a utilizar serán las peticiones síncronas.

Ahora, debemos identificar qué **tipos de conector** serían adecuados para este patrón. Sabiendo que hemos optado por una arquitectura distribuida, la elección se simplifica: los servicios pueden estar desplegados en máquinas distintas, por tanto el paso de mensajes será a través de la red.

Conociendo esto, en lugar de recurrir a la taxonomía que lista [13], optamos por consultar las estrategias de comunicación habituales para sistemas distribuidos descritas en [17]. Se trata de cuatro mecanismos distintos: Invocación a métodos remotos (*Remote Procedure Call*), APIs REST, consultas con GraphQL o *brokers* de mensajería. Tuvimos que evaluarlos mediante un análisis de *trade-offs* para determinar las ventajas y desventajas de cada uno.

Smart endpoints, dumb pipes: <https://simplicable.com/new/smart-endpoints-and-dumb-pipes>

Invocación de métodos remotos o (*Remote Procedure Call*): Esta patrón se basa en el estilo cliente-servidor. Un servidor expone una serie de funciones que el cliente puede invocar mediante peticiones a través de la red. Estas peticiones incluyen el nombre de la función a ejecutar y sus parámetros. Al finalizar la ejecución, el servidor puede devolver su resultado, si lo hubiera. Existen varios protocolos que implementan este mecanismo como gRPC o SOAP.

Una evolución de RPC suele emplearse en la programación orientada a objetos: el paradigma de **objetos distribuidos**. [18] En este caso, el programa cliente puede interactuar con objetos en servidores remotos como si fueran locales. Esta interacción se realiza a través de objetos que actúan como *proxies*, abstrayendo de la llamada al servidor.

Los *proxies* ofrecen una interfaz para que el cliente invoque sus métodos localmente. Internamente, estos métodos realizan una llamada al servicio remoto donde se encuentra el objeto realmente. El servidor remoto procesa la petición y nos devolverá un resultado. Así, abstraen al cliente de todo este proceso de comunicación. En la figura 3.8 tenemos un esquema de este mecanismo.

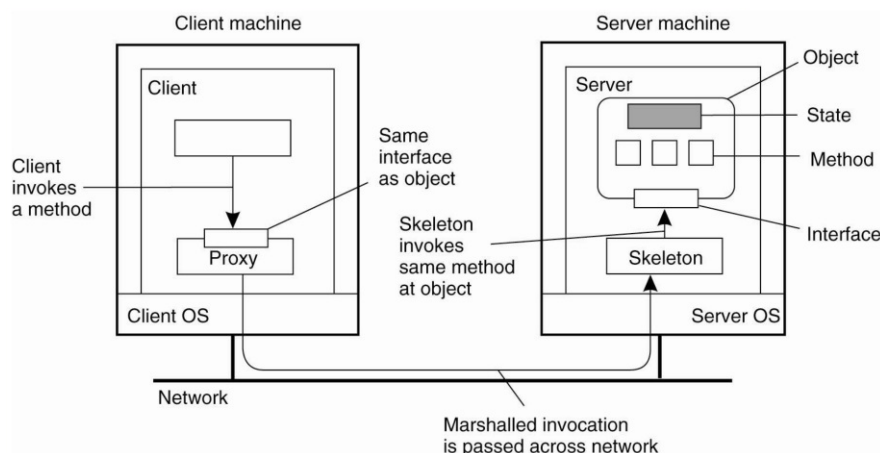


Figura 3.8: Funcionamiento del sistema de objetos distribuidos. [18]

■ Ventajas:

- Permite distribuir la carga de procesamiento del sistema. Esto puede ayudar para escalar la aplicación.
- Abstrae al cliente de la interacción con un servidor remoto. Le resulta prácticamente indistinguible de un objeto local.
- Los *proxies* o (*stubs* en la terminología de RPC) suelen generarse a partir de un contrato que define qué operaciones ofrecen estos objetos. Por ejemplo: SOAP con WDSL, gRPC; o en el caso de objetos distribuidos, Java RMI. ¿Y la ventaja?

■ **Desventajas:**

- No se puede abstraer completamente al cliente de las llamadas a través de la red. Pueden darse errores que no ocurrirían durante una invocación de un método sobre un objeto local. Por ejemplo, que el servidor no esté disponible. [19]
- Dificulta la integración con otras aplicaciones. Cada servicio ofrece sus propias funciones distintas. No están estandarizadas.
- Si adoptamos sistemas como Java RMI, nuestro sistema se acopla a esa tecnología concreta. [17]. Nos quita flexibilidad en cuanto a qué otras tecnologías podemos utilizar en nuestra arquitectura.
- El cliente debe actualizarse y recompilarse con cada cambio en el esquema del servidor. Esto puede ser problemático para casos donde tenemos que desplegar una actualización para que nuestros clientes puedan continuar utilizando la aplicación.

Representational State Transfer (REST): Se basa también en el estilo arquitectónico cliente-servidor, pero con ciertas restricciones adicionales. [8] Su concepto principal son los **recursos**: cualquier elemento sobre el que la API pueda ofrecernos información; y que pueda tener asociado un identificador único (una URI). [20] Por ejemplo, podrían ser las entidades del dominio que gestiona nuestro servicio: usuarios, mediciones de temperaturas...

Las acciones que podemos ejecutar sobre los recursos (leer, crear, actualizar, ...) las define el protocolo de comunicación sobre el que se implemente. Gracias a esto, la API que pueden ofrecer los servicios REST es común. Solo cambia el "esquema de los datos", los tipos de recursos que ofrecen. Esto facilita enormemente la integración con otros servicios. [21] La implementación más habitual es sobre el protocolo HTTP. Define métodos estandarizados como *GET* para las lecturas, *PUT* para las actualizaciones, etc.

■ **Ventajas:**

- **Stateless:** El servidor no mantiene el estado de la sesión del cliente. Esto permite que cada petición sea independiente de las demás.
- **Escalable:** Como las sesiones deben ser *stateless*, podremos replicar nuestro servicio y que distintas instancias puedan atender las peticiones que surjan durante una misma sesión.
- **API Sencilla:** Solo hay que implementar unos pocos métodos estándar para interactuar con la API.
- **Comunicación síncrona:** Es el mecanismo ideal para comunicaciones síncronas, donde el cliente requiere la respuesta del servicio para poder continuar con su procesamiento. También podemos dar soporte a para comunicaciones *fire and forget*, donde el cliente envía un mensaje y no espera ninguna respuesta a su petición.

- **Interoperabilidad:** Ampliamente utilizado en servicios de Internet. Es ideal para que clientes externos contacten con nuestro sistema mediante peticiones síncronas. [17]
- **Generación de clientes:** Para facilitar la comunicación con APIs REST, podemos generar librerías cliente utilizando el estándar OpenAPI. Lo explicaremos con más detalle en la sección 3.2.3.

■ **Desventajas:**

- **Dirigida:** Necesitamos conocer de antemano la ubicación del servidor al que queremos hacer una petición.
- **Rendimiento:** El rendimiento es peor comparado con mecanismos RPC. El tamaño de un mensaje HTTP serializado en XML o JSON es mayor que si estuviera en un formato binario.
- **API Sencilla:** También es una desventaja. Hay operaciones complejas que pueden ser difíciles de representar con los métodos ofrecidos por el protocolo de comunicación. Pueden requerir más tiempo de diseño, o incluso, ser implementados siguiendo el patrón RPC.

GraphQL² AMPLIAR: Se trata de un protocolo de consultas. Permite a los clientes ejecutar consultas personalizadas sobre los datos de un servidor. No necesitan de lógica específica para ejecutarla. De esta forma, el cliente puede obtener toda la información que necesita. Reduce el número de peticiones ejecutadas. También evita traerse datos innecesarios.

■ **Ventajas:**

- **Ideal para móviles:** Gracias a que reduce la cantidad de llamadas, es ideal para entornos donde queremos optimizar el uso de red.
- **Rendimiento:** Ofrece un mayor rendimiento comparado con otras alternativas que no ofrezcan un endpoint ya implementado. Y debemos obtener la misma información por composición, haciendo varias llamadas.

■ **Desventajas:**

- **Solo permite lecturas:** Es un lenguaje de consultas. No tiene comandos que permita escrituras.
- **Solo permite lecturas síncronas:**
- **Exponemos datos a la red:**
- **Problemas de rendimiento:** El cliente puede hacer consultas muy pesadas que penalicen el rendimiento de la base de datos sobre la que opera nuestro servicio.

Brokers de mensajería: Es un mecanismo de **comunicación asíncrona** muy popular. Sobre todo en arquitecturas basadas en eventos. Contamos con un servicio que actúa como intermediario, el *broker*. Este gestiona la comunicación entre los servicios del sistema. [17] Hay varias estrategias de comunicación posibles: colas de trabajo, *publish-suscribe*, híbrida...

²Página oficial: <https://graphql.org/>

Tomemos por ejemplo las **colas de trabajo**. [22] Es una estrategia para implementar comunicaciones asíncronas dirigidas. Nos permiten desacoplar la comunicación entre componentes usando colas de mensajería como intermediarias. Para ello, un servicio, el productor, publica mensajes en la cola. Estos mensajes representan peticiones de trabajo que pueden ser costosas de procesar. Un servicio, el trabajador, estará la escucha de los mensajes que llegan y los irá consumiendo. Estos mensajes se procesan siguiendo un orden FIFO (*first in, first out*). En la figura 3.9 mostramos un ejemplo con dos consumidores, a la escucha de la misma cola.

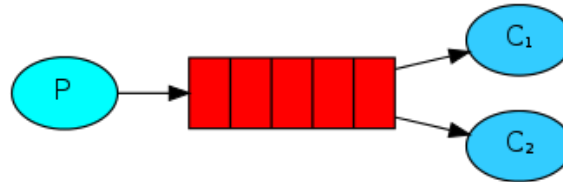


Figura 3.9: Representación de las colas de trabajo. Ejemplo de comunicación comunicación asíncrona dirigida.³

Otra estrategia posible es *publish-subscribe*: sirve para implementar comunicación *multicast*. Se basa en el uso de **temas** o **topics**: categorías de mensajes que pueden resultar de interés. Un servicio (el productor) envía un mensaje al *broker*, indicando que pertenece a un tema determinado. El *broker* recibe el mensaje y se encarga de reenviarlo a todos los servicios suscritos a este tema en concreto. [23] En la figura 3.10 tenemos un ejemplo de esta estrategia.

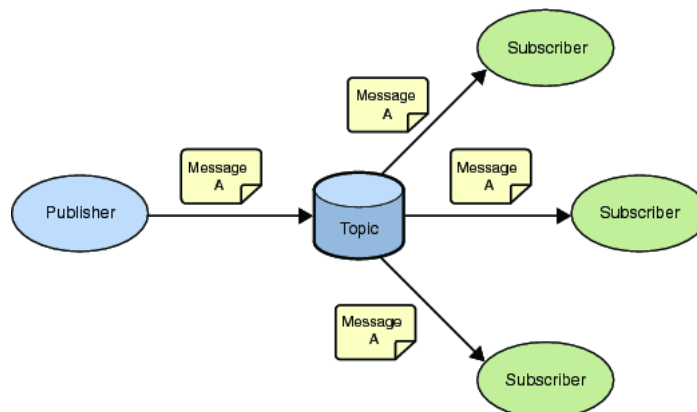


Figura 3.10: Estrategia *publish/subscribe*: el *broker* actúa como intermediario en la comunicación *multicast*. Imagen obtenida de⁴

La mayor ventaja de este estilo de comunicación es el **desacoplamiento** entre los servicios. [24] Ninguno de ellos necesita conocer detalles sobre cómo están desplegado los otros: su dirección, el número de instancias, si están activos en este momento, etc. Solo necesitan conocer el formato de los mensajes y la dirección del *broker* para enviarlos o recibirlos.

■ Ventajas:

- **Comunicación asíncrona:** El servicio no necesita quedarse a la espera de una respuesta del servidor. Puede procesar otras operaciones hasta que se le notifique del resultado, si lo hubiera.

³Imagen obtenida de: <https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html>

⁴Java Messaging Service: https://docs.oracle.com/cd/E19509-01/820-5892/ref_jms/index.html

- **Desacoplamiento de los servicios:** Ni los productores ni los consumidores necesitan conocer el origen o destino de sus mensajes. Solo su formato y la dirección del *broker*.
- **Envío garantizado de mensajes:** El *broker* garantiza que el mensaje será entregado *al menos* una vez al consumidor. Reintentará el reenvío hasta que se confirme su recepción.

■ **Desventajas:**

- **Requisitos de infraestructura:** Utilizar un *broker* de mensajería puede incrementar la dificultad de nuestros despliegues. El *broker* puede convertirse en un punto de fallo. Para operar de forma fiable, estos sistemas requieren de replicación. [17]
- **Envío garantizado de mensajes:** Para poder garantizar el envío de un mensaje, el *broker* puede recurrir a reenviarlo. Debemos diseñar nuestros sistemas de forma que estos mensajes duplicados sean descartados si ya han sido procesados.

En la tabla 3.1 presentamos un resumen de esta comparativa:

	RPC	REST	GraphQL	Broker mensajería
Tipo de comunicación entre componentes	Directa	Directa	Directa	Directa y <i>Multicast</i>
Acoplamiento entre componentes	Alto	Medio	Alto	Bajo
Interoperabilidad	Baja	Alta	Alta	Alta ⁵
Comandos de lectura	Sí	Sí	Sí	Sí
Comandos de escritura	Si	Sí	No	Sí
Comunicación síncrona	Sí	Sí	Sí	No
Comunicación asíncrona	No	Sí	No	Si

Tabla 3.1: Comparativa de los mecanismos de comunicación.

Ahora analizaremos qué protocolo elegimos para cada mecanismo de comunicación.

Peticiones síncronas

De estas cuatro opciones, podemos descartar inmediatamente la opción de GraphQL. Se trata de un conector más orientado a las consultas de datos. En nuestro caso, necesitamos ejecutar también escrituras de los valores de las propiedades. Aunque podría ser interesante para consultas más avanzadas, utilizar dos protocolos de comunicación en paralelo aumentaría la complejidad de la arquitectura.

También optamos por descartar el *broker* de mensajería. Como requerimos de comunicación directa, nos convenía que esta fuera síncrona. Para obtener propiedades del conocimiento, resultaba más sencillo de implementar mediante comunicación síncrona.

Finalmente, hay que tener en cuenta que una de nuestras prioridades es la **interoperabilidad**: es una API expuesta "hacia fuera", hacia una capa más externa; prima por

⁵Depende de si tenemos control sobre los componentes que queremos integrar.

tanto la compatibilidad con cualquier tipo de cliente. Descartamos entonces RPC, dado que nos acoplaría a una tecnología concreta y a APIs no estándares.

Terminamos por tanto decantándonos por implementar la comunicación utilizando un conector REST sobre HTTP. Implementaremos ambas funciones mediante *endpoints* HTTP. Su especificación se detalla a continuación en las tablas 3.2 y 3.3.

Operación HTTP	GET	Ruta	property / {propertyName}
Descripción	Devuelve el valor de la propiedad, si existe.		
Parámetros	propertyName	El nombre de la propiedad que deseamos obtener. Se lee a partir de la ruta de la petición.	
Respuestas posibles	Código 200 (Ok)	La propiedad se ha encontrado. Incluye un payload con el siguiente esquema: <ul style="list-style-type: none">Value: Valor de la propiedad serializado en JSON.LastModification: Fecha y hora de la última modificación de esta propiedad.	
	Código 400 (Bad request)	La petición está mal formada, no es acuerdo al contrato.	
	Código 404 (Not found)	No se ha encontrado ninguna propiedad con el nombre proporcionado.	
Ejemplo	Petición para obtener la propiedad currentTemperature: Request: HTTP GET property/currentTemperature Response: 200 Ok { value: { "Value":16.79, "Unit": 1, // Celsius "ProbeId":"c02234d3-329c-4b4d-ae0-d220dc25276b", "DateTime":"2022-01-15T18:19:38.5231231Z" }, lastModification: "2022-01-15T18:19:39.123213Z" }		

Tabla 3.2: Especificación de la operación para obtener una propiedad del servicio de conocimiento.

Una vez definida la interfaz que expondrá el servicio de conocimiento, nos queda definir cómo se contactará desde el servicio de monitorización. ¿Implementamos las llamadas manualmente con un cliente HTTP? Aunque no sería muy complicado, tendríamos que mantenerlo manualmente cuando evolucione el sistema. Optamos entonces por una alternativa: el estándar OpenAPI.

3.2.3. Open API

Operación HTTP	PUT	Ruta	property/{propertyName}
Descripción	Actualiza (o crea, si no existe) el valor de la propiedad con el nombre dado.		
Parámetros	propertyName	El nombre de la propiedad que deseamos crear o actualizar. Se lee a partir de la ruta de la petición.	
	SetPropertyDTO	Un DTO que contiene el valor a asignar en la propiedad serializado en JSON. El DTO se encuentra en el cuerpo de la petición.	
Respuestas posibles	Código 204 (No content)	La propiedad se ha creado o actualizado correctamente. No incluye <i>payload</i> en el cuerpo de la respuesta.	
	Código 400 (Bad request)	La petición está mal formada, no es acuerdo al contrato.	
Ejemplo	Petición para actualizar la propiedad <i>currentTemperature</i> con una medición de un termómetro: Request: HTTP PUT property/currentTemperature { value: { "Value":16.79, "Unit": 1, // Celsius "ProbeId":"c02234d3-329c-4b4d-ae0-d220dc25276b", "DateTime":"2022-01-15T18:19:38.5231231Z" } } Response: 204 (No content)		

Tabla 3.3: Especificación de la operación para actualizar o crear una propiedad del servicio de conocimiento.

OpenAPI es un lenguaje estándar para describir APIs RESTful. Nos permite describir de forma estructurada las operaciones que ofrece un servicio HTTP, manteniéndose agnóstico a su implementación. Esta descripción ayuda tanto a humanos como a computadoras a descubrir y utilizar las funcionalidades de la API. La OpenAPI Initiative (OAI) dirige el proyecto bajo el manto de la *Linux Foundation*.



Un documento OpenAPI habitual documenta el funcionamiento de la API y el conjunto de recursos que la componen. Describe las operaciones HTTP que podemos ejecutar sobre estos recursos, incluyendo las estructuras de datos que recibe o envía y los códigos de respuesta. Estos códigos indican al cliente el resultado de la ejecución de la operación. [25] Más adelante mostraremos un ejemplo, con el [fragmento 3.2](#).

La especificación puede escribirse manualmente o puede generarse a partir de una implementación existente. Así, podemos desarrollar nuestro servicio en un determinado lenguaje y obtener su descripción en OpenAPI. Podemos aprovecharla en varios ámbitos del desarrollo, gracias a la gran variedad de herramientas existentes: generación de

documentación, generación de casos de prueba, identificar cambios incompatibles, etc. [26]

Uno de los casos de uso más interesantes es la generación de código a partir de la definición. Existen una serie de generadores⁶ capaces de generar clientes o servidores conformes a la especificación. Ofrecen soporte a una gran variedad de lenguajes: Java, C#, JavaScript... En el caso de cliente, actúa como un proxy que nos abstrae de la lógica de comunicación con el servidor, similar a lo descrito en el apartado de RPC.

Para el desarrollo de este trabajo, nos interesaba especialmente debido a las diferencias tecnológicas existentes: el bucle MAPE-K original estaba desarrollado en Java, pero el prototipo se desarrolló con el lenguaje C# junto con el framework ASP.NET Core. Se tomó esta decisión para reducir el tiempo de aprendizaje y centrar los esfuerzos en la definición de la arquitectura del sistema.

Gracias a la generación de código, pudimos obtener la especificación de los servicios desarrollados en ASP.NET Core, y generar clientes o servidores en cualquier lenguaje soportado, Java incluido. El bucle MAPE-K original después podría ser refactorizado usando este código autogenerado.

Ejemplo de uso

A continuación explicaremos brevemente cómo utilizamos OpenAPI para documentar nuestras APIs y generar la especificación estas. Para ello, continuaremos con el ejemplo del servicio de conocimiento que hemos descrito a lo largo de este capítulo. Vamos a centrarnos en la implementación de la operación para obtener una propiedad del conocimiento, que describimos en la tabla 3.2.

En el [fragmento 3.1](#), podemos observar que se trata de un método C# llamado *GetProperty*. Su implementación es sencilla: busca en un diccionario la propiedad cuyo nombre se le pasa por parámetro. En caso de encontrarla, devuelve su valor con un código 200 OK. En caso contrario, devuelve un código de error que describe qué ha ocurrido exactamente (llamada incorrecta o no se ha encontrado la propiedad).

Aparte de la implementación, podemos comprobar que el método se ha decorado con una serie de comentarios (líneas 1-8) y atributos (10-12). Esta documentación describe qué hace el método, sus entradas y posibles respuestas. OpenAPI es capaz de utilizar estos elementos opcionales para generar una especificación más completa. Por tanto, resulta muy recomendable utilizarlos.

```
1 /// <summary>
2 ///     Gets a property given its name.
3 /// </summary>
4 /// <param name="propertyName"> The name of the property to find. </param>
5 /// <returns> An IActionResult with result of the query. </returns>
6 /// <response code="200"> The property was found. Returns the value of the
7 ///     property. </response>
8 /// <response code="404"> The property was not found. </response>
9 /// <response code="400"> There was an error with the provided arguments. </
10 ///     response>
11 [HttpGet(" {propertyName} ")]
12 [ProducesResponseType(typeof(PropertyDTO), StatusCodes.Status200OK)]
13 [ProducesResponseType(StatusCodes.Status404NotFound)]
14 [ProducesResponseType(StatusCodes.Status400BadRequest)]
15 public IActionResult GetProperty([FromRoute] string propertyName)
16 {
17     if (string.IsNullOrEmpty(propertyName))
18     {
```

⁶<https://github.com/OpenAPITools/openapi-generator>


```
17         return BadRequest();
18     }
19
20     bool foundProperty = properties.TryGetValue(propertyName, out PropertyDTO
21         property);
22
23     if (!foundProperty)
24     {
25         return NotFound();
26     }
27
28     return Ok(property);
29 }
```

Listing 3.1: Implementación del método GetProperty decorado para generar la especificación OpenAPI.

Haciendo uso de las librerías de OpenAPI, generamos la especificación a partir del servicio de conocimiento. En el [fragmento 3.2](#), podemos ver cómo se describe la operación en este estándar:

```
1  "paths": {
2    "/Property/{propertyName}": {
3      "get": {
4        "tags": [
5          "Property"
6        ],
7        "summary": "Gets a property given its name.",
8        "parameters": [
9          {
10             "name": "propertyName",
11             "in": "path",
12             "description": "The name of the property to find.",
13             "required": true,
14             "schema": {
15               "type": "string"
16             }
17           }
18         ],
19         "responses": {
20           "200": {
21             "description": "The property was found. Returns the value of the
22               property.",
23             "content": {
24               "application/json": {
25                 "schema": {
26                   "$ref": "#/components/schemas/PropertyDTO"
27                 }
28               }
29             }
30           },
31           "404": {
32             "description": "The property was not found.",
33           },
34           "400": {
35             "description": "There was an error with the provided arguments.",
36           }
37         }
38       }
39     }
40   }
```

Listing 3.2: Especificación OpenAPI del método para obtener una propiedad del conocimiento (GetProperty).

Podemos apreciar que en la ruta (*/Property/{propertyName}*) está disponible una operación de tipo *get* y que acepta determinados parámetros y ofrece unas posibles respuestas. Aparece una referencia a otro esquema (línea 25), que representa la estructura de la respuesta en ese caso concreto. También aparecen los comentarios opcionales que indicamos en el [fragmento 3.1](#). Encontramos grandes similitudes con la especificación presentada en la [tabla 3.2](#).

Los convenios de los generadores de código de OpenAPI pueden no ser de nuestro agrado. Por ejemplo, pueden resultar muy verbosos o puede resultar muy pesado trabajar con DTOs directamente. Por suerte, tenemos dos opciones para solventar esto: Modificar las plantillas de generación de código. Al ser de código abierto, podríamos modificar las existentes o crear nuestras propias plantillas con nuestros propios convenios.

Otra opción, más fácil de implementar, es desarrollar código por encima del API Client Generado. Es el caso del servicio de Análisis. Como trabajar con DTOs directamente se hacía muy pesado, optamos por implementar un 'system configuration request' builder. Esto nos permitía configurar la petición de una forma más descriptiva para el usuario:

```

1 var changeRequests = new List<ServiceConfigurationDTO>
2 {
3     new()
4     {
5         ServiceName = ClimatisationAirConditionerConstants.AppName,
6         IsDeployed = true,
7         ConfigurationProperties = new List<ConfigurationProperty>()
8         {
9             new()
10            {
11                Name = ClimatisationAirConditionerConstants.Configuration.Mode,
12                Value = AirConditioningMode.Cooling.ToString(),
13            },
14        },
15    },
16 };
17
18 var symptoms = new List<SymptomDTO> { new(SymptomName, "true") };
19
20 var systemConfigurationChangeRequest = new SystemConfigurationChangeRequestDTO
21 ()
22 {
23     ServiceConfiguration = changeRequests,
24     Symptoms = symptoms,
25     Timestamp = DateTime.UtcNow,
26 };
27
28 await _systemApi.SystemRequestChangePostAsync(
29     systemConfigurationChangeRequest,
30     CancellationToken.None);

```

Listing 3.3: Implementación de la misma petición siguiendo el patrón *builder*.

```

1 await _systemService.RequestChangeAsync(changeRequest =>
2 {
3     changeRequest
4         .ForSymptom(TemperatureGreaterThanHotThreshold)
5         .WithService(ClimatisationAirConditionerConstants.AppName, service =>
6         {
7             service.MustBePresent()
8             .WithParameter(
9                 ClimatisationAirConditionerConstants.Configuration.Mode,
10                 AirConditioningMode.Cooling.ToString());

```

$$\begin{array}{l|l} 11 & \} \} ; \\ 12 & \} \} ; \end{array}$$

Listing 3.4: Implementación de la misma petición siguiendo el patrón *builder*.

Finalmente, la arquitectura del conector que emplearemos para implementar las peticiones aparece en la figura 3.11. La figura muestra como el servicio de monitorización contacta al de conocimiento para asignarle un valor a la propiedad *Temp*.

El conector, delimitado por una línea discontinua roja, está compuesto por dos elementos: una API REST y un cliente. Los otros dos grupos de elementos representan los procesos de los servicios de monitorización y conocimiento. El servicio de monitorización se comunica a con la API través del API Client, que está en su proceso actuando como *proxy*.

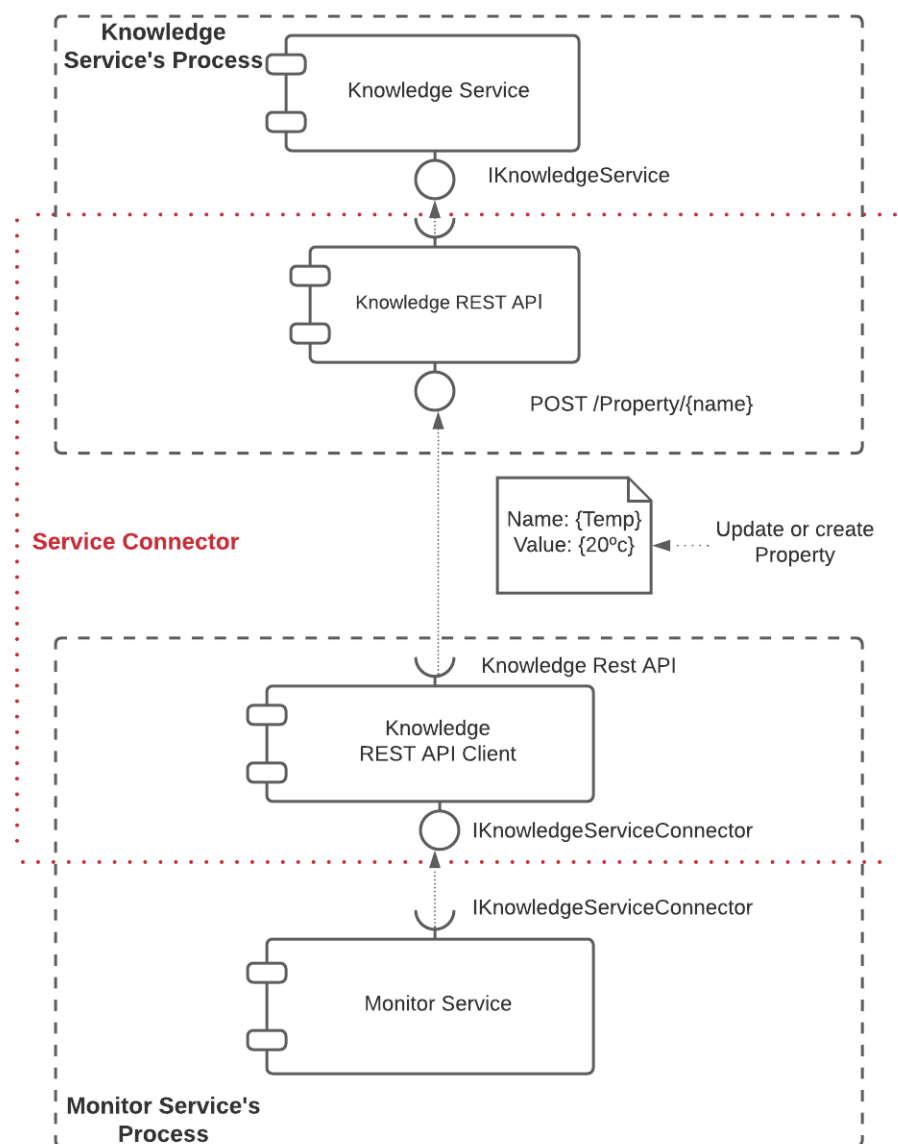


Figura 3.11: Diseño del conector usando implementación Cliente - Servidor

3.2.4. Notificaciones

El siguiente mecanismo de comunicación a definir son las notificaciones. Recordemos que esta comunicación es desde un servicio a todos los que se encuentren en la capa superior (*multicast*). No debe estar acoplada a ningún servicio concreto. Potencialmente, todos deberían recibir el mensaje y decidir si procesarlo o no.

Como ejemplo, tomaremos la comunicación entre el servicio de conocimiento y los servicios en la capa superior (el nivel del bucle). Cada vez que se modifique una propiedad o una configuración de un servicio, el conocimiento emitirá un evento notificando del cambio a la capa superior. Así, por ejemplo, el análisis sabrá que debe reevaluar las reglas de adaptación.

Sabiendo esto, podemos descartar de entrada GraphQL. Es un protocolo basado en lecturas. Como el objetivo es enviar información a otros servicios, no nos sirve. Respecto a RPC y REST, tampoco nos sirven, no tienen un buen soporte de multicast. Además de que tenemos el requisito de bajo acoplamiento.

Por tanto, optamos por implementarlo usando un *broker* de mensajería. Concretamente, siguiendo el patrón *publish-subscribe*. El servicio de conocimiento publicará el evento a través del *broker* de mensajería. Este evento tendrá un *topic* asociado. Todos los servicios interesados deberán suscribirse a este *topic*. El *broker* reenviará el mensaje a una cola específica para cada uno de los servicios suscritos al tema. Así podrán procesarlo cuando puedan, de forma asíncrona.

De esta manera logramos el desacoplamiento de los componentes y permitimos el procesamiento asíncrono de estos eventos.

El evento que publicaría el módulo de conocimiento cuando cambia una propiedad podría ser como el siguiente:

```
{ "$type": "Knowledge.Contracts.IntegrationEvents.PropertyChangedIntegrationEvent", Knowledge
```

Los eventos incluirán la información mínima indispensable. En este caso, el nombre de la propiedad que ha cambiado. Esta decisión la tomamos así debido a que es una comunicación asíncrona. Si el evento incluyera el valor de la propiedad y se procesa mucho más tarde, podría derivar en adaptaciones. Así evitamos una adaptación incorrecta del sistema. Para evitarlo, incluyendo solo el nombre de la propiedad, obligamos a las reglas a que soliciten el valor de la propiedad en el momento en que se evalúen. Así siempre se ejecutarán con la información actualizada.

Aunque potencialmente otros servicios podrían suscribirse a estos cambios, vamos a describir solo la suscripción del módulo de análisis.

Para definir esta comunicación investigamos si había algún estándar equivalente a OpenAPI. Y así es. Se llama AsyncAPI. Por desgracia, todavía está en fases preliminares de desarrollo, y no tiene la implantación que ha tenido OpenAPI. Por ejemplo, no tiene el nivel de generación de código que tiene el primero.

Aun así, lo utilizaremos para describir los mensajes en un formato estándar.

CAPÍTULO 4

Caso de estudio: Sistema de climatización

Para verificar la arquitectura definida, decidimos implementar un pequeño sistema autoadaptativo. Se trata de un sistema de climatización, que gestiona la temperatura de una habitación. Para ello, dispondremos de un aire acondicionado, que calentará o enfriará la habitación según corresponda.

4.1 Análisis

El primer paso es capturar los requisitos del sistema a implementar. Cómo hemos comentado, queremos desarrollar un sistema de climatización. Este sistema regulará la temperatura de una habitación mediante el uso de un aparato de aire acondicionado.

El aparato de aire acondicionado ofrece tres modos de funcionamiento: un modo para calentar la estancia, otro para enfriarla, y un estado neutral (apagado). Además, lo hemos dotado con un termómetro interno que nos reporta la temperatura periódicamente.

Para poder climatizar la habitación, necesitamos que el usuario defina su temperatura objetivo: la temperatura de confort. Cambios en la temperatura deberán activar o desactivar el aparato para mantenerla.

Además, nos interesa evitar que el aire acondicionado se encienda y se apague constantemente cuando se alcance o sobrepase esta temperatura. Por ello, definimos unas temperaturas umbrales, tanto de frío como de calor, a partir de las cuales se encenderá el aparato.

4.2 Diseño

Del análisis anterior ya podemos deducir la existencia de dos componentes: un aparato de aire acondicionado (el sistema gestionado) y un termómetro (la sonda). Aparte de ellos, deberemos implementar la infraestructura necesaria para comunicarse con nuestro bucle MAPE-K: monitores, módulos de reglas y efectores que nos permitan interactuar con el sistema manejado.

Para describir el diseño usaremos la notación de sistemas autoadaptativos descrita en [7].

4.2.1. Son das:

Para implementar el sistema, requerimos de las siguientes sondas:

Sonda:	<i>thermometer</i>
Descripción:	Reporta la temperatura actual de la habitación (en °c).
Monitor:	<i>Climatisation.Monitor</i>
Datos:	<i>temperature</i>
Sonda:	<i>airconditioner-mode-changed-probe</i>
Descripción:	Reporta el modo de funcionamiento del aire acondicionado cuando este cambia.
Monitor:	<i>Climatisation.Monitor</i>
Datos:	<i>airconditioner-mode</i>
Sonda:	<i>airconditioner-adaption-loop-registration</i>
Descripción:	Cuando arranca el servicio de aire acondicionado, registra la configuración inicial del sistema.
Monitor:	<i>Climatisation.Monitor</i>
Datos:	<i>airconditioner.is-deployed, airconditioner-mode, target-temperature, cold-temperature-threshold, hot-temperature-threshold</i>

Tabla 4.1: Son das del sistema de climatización.

4.2.2. Propiedades de adaptación:

También podemos deducir cuáles son nuestras propiedades de adaptación:

Propiedad:	<i>temperature</i>
Descripción:	Representa la temperatura actual de la habitación (en °C).
Tipo de dato:	<i>float</i>
Propiedad:	<i>target-temperature</i>
Descripción:	La temperatura de confort definida por el usuario. El sistema deberá adaptarse para alcanzarla.
Tipo de dato:	<i>float</i>
Propiedad:	<i>cold-temperature-threshold</i>
Descripción:	La temperatura umbral de frío (en °c). Si la temperatura baja por debajo de este umbral, deberá calentarse la habitación.
Tipo de dato:	<i>float</i>
Propiedad:	<i>hot-temperature-threshold</i>
Descripción:	La temperatura umbral de calor (en °c). Si la temperatura sube por encima de este umbral, deberá enfriarse la habitación.
Tipo de dato:	<i>float</i>
Propiedad:	<i>airconditioner.is-deployed</i>
Descripción:	Indica si el servicio de aire acondicionado está desplegado y en funcionamiento.
Tipo de dato:	<i>bool</i>
Propiedad:	<i>airconditioner-mode</i>
Descripción:	Representa el modo de operación actual del aire acondicionado: <i>Off</i> = 0, <i>Cooling</i> = 1, <i>Heating</i> = 2
Tipo de dato:	Enumerado

Tabla 4.2: Propiedades de adaptación del sistema de climatización.

4.2.3. Monitores:

Necesitaremos definir varios monitores para capturar los datos de las sondas. En algunos casos, para evitar falsos positivos, y que se lleve a cabo adaptaciones provocadas por errores de medición, deberemos filtrar estos datos.

Por ejemplo, en el monitor de las temperaturas, *climatisation.monitor.temperature*. Como en el ejemplo trabajamos con un aire acondicionado ficticio, le hemos establecido un margen de error grande: Si la nueva medida de temperatura está a 5°C de diferencia o más, y hay menos de un minuto de diferencia entre ellas; la descartaremos. De esta forma, evitamos que el aire acondicionado se active o desactive por un error de medición.

Monitor:	<i>climatisation.monitor.temperature</i>
Descripción:	Recibe los reportes de temperatura de los termómetros. También filtra estos datos para detectar casos donde se sospecha un error de lectura.
Afecta a propiedades de adaptación:	<i>temperature</i>
Acciones:	SI $ new_temperature - temperature \leq 5.0$ O $request.DateTime - previousMeasurement.DateTime > 60s$ ACTUALIZA-KNOWLEDGE <i>temperature = new-temperature</i>
Monitor:	<i>climatisation.monitor.configuration</i>
Descripción:	Recibe la configuración del aire acondicionado y la registra en el <i>knowledge</i> .
Afecta a propiedades de adaptación:	<i>airconditioner.is-deployed, airconditioner-mode, target-temperature, cold-temperature-threshold, hot-temperature-threshold</i>
Acciones:	SI <i>property != new-value</i> ACTUALIZA-KNOWLEDGE <i>property = new-value</i>

Tabla 4.3: Monitores del bucle MAPE-K del sistema de climatización.

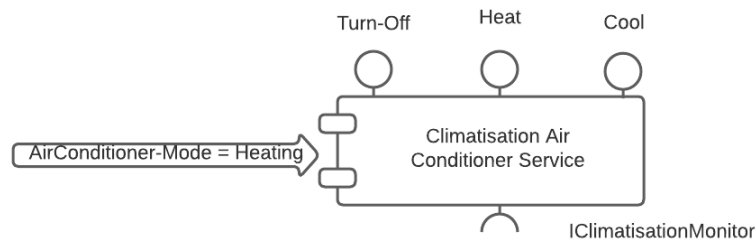
4.2.4. Reglas de adaptación

En base a cambios de la temperatura local, deberemos decidir si es necesario llevar a cabo una acción correctiva. Por ejemplo, que si la temperatura es inferior al umbral de temperatura fría, el aparato se enciende en modo calentador. Para ello, deberemos implementar un servicio de reglas (*Climatisation.Rules.Service*). En él, incluiremos una serie de reglas que se disparen cuando cambie una de nuestras propiedades de adaptación. En este caso, la temperatura.

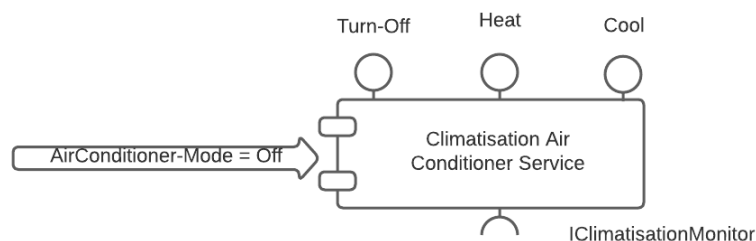
Como comentamos en el capítulo anterior, en nuestro ejemplo de bucle MAPE-K, nos limitamos a implementar las adaptaciones de tipo set-parameter. Por tanto, no tendremos reglas de despliegue o de binding.

En la tabla 4.4 definimos las cuatro reglas necesarias:

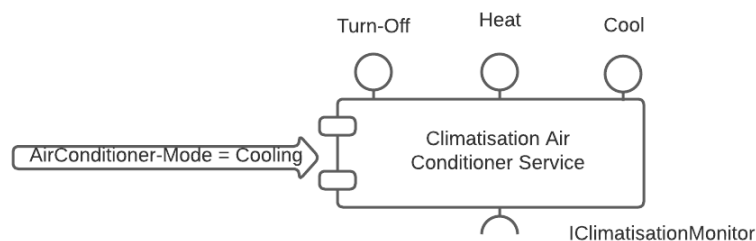
Regla:	<i>EnableAirConditionerHeatingModeWhenColdTemperatureThresholdExceeded</i>
Descripción:	Activa el aire acondicionado en modo calefacción cuando la temperatura sea inferior al umbral de frío.
Condición:	<i>airconditioner-mode != Heating AND temperature <= cold-temperature-threshold</i>
Cuerpo:	



Regla: *DisableAirConditionerWhenHeatingModeEnabledAndTargetTemperatureAchieved*
Descripción: Apaga el aire acondicionado cuando el modo calefacción está activo y se ha alcanzado la temperatura de confort.
Condición: *airconditioner-mode == Heating AND temperature >= target-temperature*
Cuerpo:



Regla: *EnableAirConditionerCoolingModeWhenTemperatureThresholdExceeded*
Descripción: Activa el aire acondicionado en modo enfriar cuando la temperatura sea superior al umbral de calor.
Condición: *airconditioner-mode != Cooling AND temperature >= hot-temperature-threshold*
Cuerpo:



Regla: *DisableAirConditionerWhenCoolingModeEnabledAndTargetTemperatureAchieved*
Descripción: Apaga el aire acondicionado cuando el modo enfriar está activo y se ha alcanzado la temperatura de confort.
Condición: *airconditioner-mode == Cooling AND temperature <= target-temperature*
Cuerpo:

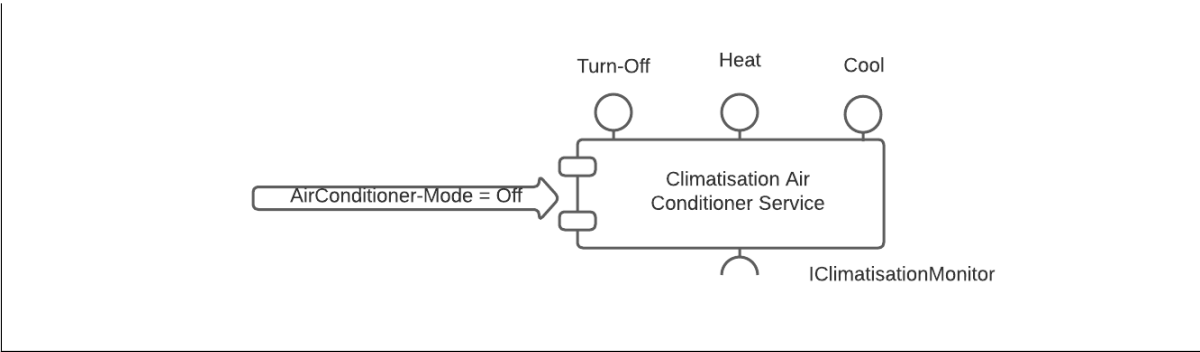


Tabla 4.4: Reglas de adaptación del sistema de climatización.

4.2.5. Efectores:

Una vez se evaluén estas reglas, solicitamos un cambio en la configuración del sistema. El módulo de planificación comprobará contra el conocimiento y el estado actual del sistema cuáles de los cambios solicitados es necesario aplicar. Si por ejemplo la propiedad ya tiene el valor solicitado, no hará falta ejecutarla.

El modulo de ejecución recibirá la petición y se la redirigirá a los efectores del sistema de climatización. En este caso, requerimos de efectores que cambien el modo del aire acondicionado según corresponda.

Efactor:	<i>airconditioner.heat</i>
Descripción:	Activa el modo calentar del aire acondicionado.
Efactor:	<i>airconditioner.cool</i>
Descripción:	Activa el modo enfriar del aire acondicionado.
Efactor:	<i>airconditioner.turn-off</i>
Descripción:	Apaga el aire acondicionado.

Tabla 4.5: Efectores del sistema de climatización.

Hecho esto, el sistema se adapta a a la nueva situación, y reportará una nueva temperatura en cuanto corresponda. La temperatura variará dependiendo de si está apagado o no.

4.2.6. Configuración del sistema

Requerimos entonces 4 servicios para implementar la solución: Servicio de aire acondicionado, monitor de climatización, el servicio de reglas y el servicio de efectores. Con ellos, podemos adaptarnos al bucle MAPE-K descrito en el capítulo

4.3 Implementación

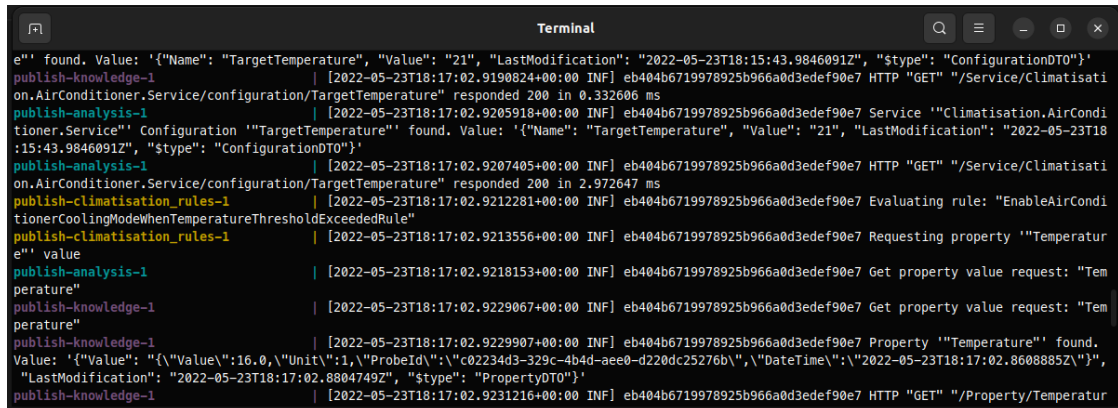
Para la implementación, hemos utilizado las mismas tecnologías que en los servicios del bucle MAPE-K: microservicios en ASP.NET Core, empaquetados en contenedores de Docker para facilitar su despliegue. Generamos los API Clients con OpenAPI y demás.

Como no disponemos de un aire acondicionado real, hemos optado por implementar uno ficticio. Cuando está apagado, la temperatura aumenta o disminuye según una configuración del fake. De esta forma, podemos simular los cambios de temperatura más rápido y ver si se aplican las adaptaciones pertinentes.

4.3.1. Telemetría

Un punto en el que queremos hacer hincapié es en la telemetría. Debido a que estamos tratando con un sistema distribuido es complicado conocer el estado del sistema en determinado momento. Especialmente en este caso, que participan más de diez servicios distintos.

Por defecto, solo contábamos con los *logs* de consola, que mostramos en la figura 4.1. Aparecen en una única ventana intercalados los registros de todos los servicios. Aunque nos pueden resultar útil, es una aproximación ineficiente y según aumente la escala de peticiones simultáneamente se volverá más difícil de interpretar.



```

e" found. Value: '{"Name": "TargetTemperature", "Value": "21", "LastModification": "2022-05-23T18:15:43.9846091Z", "type": "ConfigurationDT0"}'
publish-knowledge-1 | [2022-05-23T18:17:02.9190824+00:00 INF] eb404b6719978925b966a0d3edef90e7 HTTP "GET" "/Service/Climatisati
on.AirConditioner.Service/configuration/TargetTemperature" responded 200 in 0.332606 ms
publish-analysis-1 | [2022-05-23T18:17:02.9205918+00:00 INF] eb404b6719978925b966a0d3edef90e7 Service "Climatisation.AirCondi
tioner.Service" Configuration "TargetTemperature" found. Value: '{"Name": "TargetTemperature", "Value": "21", "LastModification": "2022-05-23T18
:15:43.9846091Z", "type": "ConfigurationDT0"}'
publish-analysis-1 | [2022-05-23T18:17:02.9207405+00:00 INF] eb404b6719978925b966a0d3edef90e7 HTTP "GET" "/Service/Climatisati
on.AirConditioner.Service/configuration/TargetTemperature" responded 200 in 2.972647 ms
publish-climatisation_rules-1 | [2022-05-23T18:17:02.9212281+00:00 INF] eb404b6719978925b966a0d3edef90e7 Evaluating rule: "EnableAirCondi
tionerCoolingModeWhenTemperatureThresholdExceededRule"
publish-climatisation_rules-1 | [2022-05-23T18:17:02.9213556+00:00 INF] eb404b6719978925b966a0d3edef90e7 Requesting property "Temperatur
e" value
publish-analysis-1 | [2022-05-23T18:17:02.9218153+00:00 INF] eb404b6719978925b966a0d3edef90e7 Get property value request: "Tem
perature"
publish-knowledge-1 | [2022-05-23T18:17:02.9229067+00:00 INF] eb404b6719978925b966a0d3edef90e7 Get property value request: "Tem
perature"
publish-knowledge-1 | [2022-05-23T18:17:02.9229907+00:00 INF] eb404b6719978925b966a0d3edef90e7 Property "Temperature" found.
Value: '{"Value": "16.0", "Unit": "1", "ProbeId": "c02234d3-329c-4b4d-ae08-d220dc5276b", "DateTime": "2022-05-23T18:17:02.8608885Z"}',
"LastModification": "2022-05-23T18:17:02.8804749Z", "type": "PropertyDT0"}'
publish-knowledge-1 | [2022-05-23T18:17:02.9231216+00:00 INF] eb404b6719978925b966a0d3edef90e7 HTTP "GET" "/Property/Temperatur

```

Figura 4.1: Extracto de *logs* de una ejecución habitual.

Por ello, para que resultara más sencillo trabajar en la implementación de los servicios y diagnosticar qué ocurre con el sistema, decidimos implementar una solución de observabilidad. La observabilidad es [27] y consta de tres partes distintas:

- **Logs:** A recording of an Event. Typically the record includes a timestamp indicating when the Event happened as well as other data that describes what happened, where it happened, etc. [28] Provide extremely fine-grained detail on a given service, but have no built-in way to provide that detail in the context of a request. [27]
- **Métricas:** Son agregados que nos permiten conocer el estado de las estancias de nuestros servicios. Records a data point, either raw measurements or predefined aggregation, as timeseries with Metadata. [28]
- **Trazas distribuidas:** Tracks the progression of a single Request, called a Trace, as it is handled by Services that make up an Application. A Distributed Trace transverse process, network and security boundaries. [28] providing visibility into the operation of your microservice architecture. It allows you to gain critical insights into the performance and status of individual services as part of a chain of requests in a way that would be difficult or time-consuming to do otherwise. Distributed tracing gives you the ability to understand exactly what a particular, individual service is doing as part of the whole, enabling you to ask and answer questions about the performance of your services and your distributed system. [27]

Para poder capturar todos estos elementos, optamos por usar el estándar OpenTelemetry. Se trata de una librería estándar utilizada para instrumentar el código de las aplicaciones. Distintas compañías del ámbito de la telemetría software ofrecen APIs que capturan el output de esta librería.

Gracias a él pudimos capturar la telemetría de la siguiente forma implementar usando tres servicios distintos:

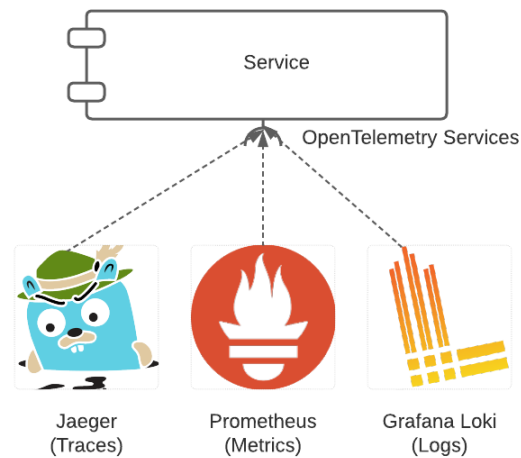


Figura 4.2: Extracto de *logs* de una ejecución habitual.

Loki: Logs

Lo primero que queremos ver es cómo mejorar nuestra estrategia de logging. Lo ideal es añadir identificadores de correlación (el traceID), que nos permita rastrear a través de los distintos servicios una misma traza. Por ejemplo, podemos filtrar a partir de ella para ver todos los detalles de los servicios que intervinieron.

Jaeger: Trazas distribuidas

Gracias a las trazas distribuidas, podemos ver todas las actividades por las que pasó una petición. En nuestro caso, podemos ver por todos los estados por los que paso.

Prometheus: Métricas

Grafana: Visualización

Desarrollamos un panel de monitorización con Grafana. Esto nos permitía consultar en un solo lugar las métricas, los logs y las trazas.

CAPÍTULO 5

Conclusions

????? ?????????????? ?????????????? ?????????????? ?????????????? ??????????????

Bibliografía

- [1] D. Garlan, S.-W. Cheng, and B. Schmerl, "Increasing System Dependability through Architecture-Based Self-Repair," in *Architecting Dependable Systems* (R. de Lemos, C. Gacek, and A. Romanovsky, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 61–89, Springer, 2003.
- [2] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw, "Engineering Self-Adaptive Systems through Feedback Loops," in *Software Engineering for Self-Adaptive Systems* (B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, eds.), pp. 48–70, Berlin, Heidelberg: Springer, 2009.
- [3] I. Corporation, "An Architectural Blueprint for Autonomic Computing," tech. rep., IBM, 2006.
- [4] J. Fons, V. Pelechano, M. Gil, and M. Albert, "Servicios adaptive-ready para la reconfiguración dinámica de arquitecturas de microservicios," in *Actas de las XVI Jornadas de Ingeniería de Ciencia e Ingeniería de Servicios, SISTEDES*, 2021.
- [5] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Transactions on Autonomous and Adaptive Systems*, vol. 1, pp. 223–259, Dec. 2006.
- [6] C. Savaglio, M. Ganzha, M. Paprzycki, C. Bădică, M. Ivanović, and G. Fortino, "Agent-based Internet of Things: State-of-the-art and research challenges," *Future Generation Computer Systems*, vol. 102, pp. 1038–1053, Jan. 2020.
- [7] J. Fons, "Especificación de sistemas auto-adaptativos," Mar. 2021.
- [8] R. N. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Jan. 2009.
- [9] N. C. Mendonça, D. Garlan, B. Schmerl, and J. Cámara, "Generality vs. reusability in architecture-based self-adaptation: The case for self-adaptive microservices," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, (Madrid Spain), pp. 1–6, ACM, Sept. 2018.
- [10] B. Foote and J. Yoder, "Big Ball of Mud," in *Fourth Conference on Patterns Languages of Programs*, (Monticello), Sept. 1997.
- [11] IEEE, ISO, and IEC, "Standard 42010-2011 - Systems and software engineering – Architecture description," tech. rep., 2011.
- [12] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, Oct. 1992.

- [13] N. R. Mehta, N. Medvidovic, and S. Phadke, "Towards a taxonomy of software connectors," in *Proceedings of the 22nd International Conference on Software Engineering, ICSE '00*, (New York, NY, USA), pp. 178–187, Association for Computing Machinery, June 2000.
- [14] R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow, "A component- and message-based architectural style for GUI software," *IEEE Transactions on Software Engineering*, vol. 22, pp. 390–406, June 1996.
- [15] "UCI Software Architecture Research - UCI Software Architecture Research: C2 Style Rules."
- [16] R. C. Martin, "Chapter 22: The Clean Architecture," in *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin Series, London, England: Prentice Hall, 2018.
- [17] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Aug. 2021.
- [18] A. S. Tanenbaum and M. van Steen, "Chapter 10: Distributed Object-Based Systems," in *Distributed Systems: Principles and Paradigms*, Pearson Prentice Hall, second ed., 2007.
- [19] P. Jausovec, "Fallacies of distributed systems," Nov. 2020.
- [20] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, May 2007.
- [21] M. Nally, "REST vs. RPC: What problems are you trying to solve with your APIs?," Oct. 2018.
- [22] G. Roy, "Chapter 6. Message patterns via exchange routing," in *RabbitMQ in Depth*, Manning Publications, Sept. 2017.
- [23] RabbitMQ, "Publish/Subscribe documentation."
- [24] J. Korab, *Understanding Message Brokers*. O'Reilly Media, June 2017.
- [25] OpenAPI_Initiative, "OpenAPI Specification v3.1.0."
- [26] D. Westerveld, "Chapter 3: OpenAPI and API Specifications," in *API Testing and Development with Postman*, Packt Publishing, May 2021.
- [27] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, "1. The Problem with Distributed Tracing," in *Distributed Tracing in Practice*, O'Reilly Media, Inc., Apr. 2020.
- [28] OpenTelemetry, "OpenTelemetry Documentation," 2022.

APÉNDICE A

Configuració del sistema

???? ????????????? ????????????? ????????????? ????????????? ?????????????

A.1 Fase d'inicialització

???? ????????????? ????????????? ????????????? ????????????? ?????????????

A.2 Identificació de dispositius

???? ????????????? ????????????? ????????????? ????????????? ?????????????

APÉNDICE B

??? ?????????????????? ???? ?

???? ????????????????? ????????????????? ????????????????? ????????????????? ?????????????????