UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

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

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Adriano Vega Llobell

Tutor: Joan Josep Fons Cors

Curso 2021-2022

Resum

????

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

Resumen

????

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

Abstract

????

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

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

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 [2, 3], 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

En la figura 1.1 tenemos una representación de la arquitectura del bucle.

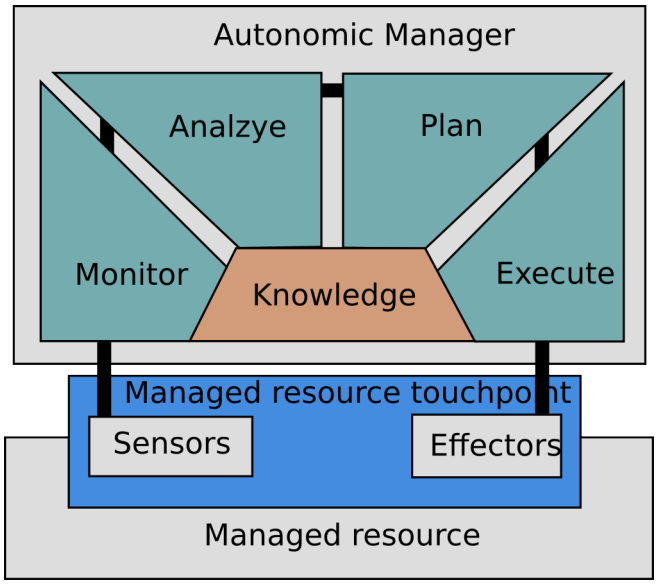

Figura 1.1: Representación del Bucle MAPE-K.[1]

El recurso manejado puede ser un sistema *hardware* o *software* cualquiera. El único requisito es que debe implementar los ***touchpoints*** (puntos de contacto?): interfaces que permiten al bucle de control obtener información del estado del sistema y cambiar su configuración en base a las políticas. Hay dos tipos de *touchpoints*: **sondas** y **efectores**.

Las sondas reportan al bucle información del estado del sistema. Puede ser cualquier tipo de métrica que queramos controlar. Por ejemplo, *health checks*, información de salud de la aplicación; propiedades del sistema que queramos controlar.

[1] Obtenido de: https://wwwvs.cs.hs-rm.de/vs-wiki/index.php/(WS12-01)_Cloud/Dokumentation

Por otro lado, los efectores, nos ayudan a modificar el estado del sistema manejado. Pueden ser ficheros de configuración, comandos, etc.

En la figura 1.1 podemos apreciar que el bucle puede dividirse en 5 componentes distintos: [2]

- **Base de conocimiento**: almacena el conocimiento relevante para la operación del bucle de control. Es tanto información del sistema como información del entorno de operación. Cada una de las claves almacenadas se conoce también como **propiedad de adaptación**.

  El conocimiento se comparte entre todos los componentes del bucle de control.

- **Monitor**: Recibe mediciones de las sondas del recurso manejado. Se encarga de recoger, agregar y filtrar estas mediciones para determinar si ha ocurrido un evento relevante que deba ser reportado. Por ejemplo, si la temperatura de una habitación supera un umbral definido por el usuario.

- **Analizador**: Conjunto de **reglas de adaptación** que se suscriben a las propiedades de adaptación. Están compuestas una condición y una acción. Cada vez que cambie alguna de las propiedades de las que dependen, se evalua su condición. Si esta se cumple, se ejecuta la acción asociada, que suele ser una propuesta de cambio en la configuración del sistema.

- **Planificador**: Si se ha llegado a ejecutar alguna regla de apdatación, el planificador recoge sus propuestas de cambio y determina las acciones necesarias para cumplir el objetivo.

- **Ejecutor**: Recibe

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
Arquitectura de la solución

En este capítulo vamos a describir la arquitectura que hemos diseñado para distribuir el bucle MAPE-K. Como partimos de un sistema existente, del cual ya conocemos sus componentes, el principal foco de este capítulo son los **conectores de *software***. Tenemos que determinar qué estrategias de comunicación utilizaremos para comunicarlos.

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.

2.1 Arquitecturas de *software*

Según [4], 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.

Conocer la arquitectura de un sistema nos permite conocerlo en profundidad. Nos permite descubrir qué **componentes** la conforman, cómo están **relacionados** y el por qué de esta composición. [5] Además, nos provee de un vocabulario común para que los *stakeholders* o interesados puedan comunicarse: ayuda a los clientes a articular sus necesidades; a los diseñadores a capturar los requisitos y definir el sistema; a los desarrolladores a comprender mejor cómo implementarlo...

Todos los sistemas *software* cuentan con una arquitectura. La diferencia radica en si esta ha sido diseñada explícitamente o ha quedado implícita en su implementación. [4] En caso de no haber sido diseñada, es probable que con el paso del tiempo el sistema vaya perdiendo su estructura, se desvirtuen las intenciones iniciales, y sea dificil de mantener. Se convierte en una "gran bola de barro". [CITATION NEEDED]

Por ello, la principal función de una buena arquitectura es dotar de estructura a nuestro sistema. [6] Una buena arquitectura nos ofrece una serie de ventajas, como facilitar su desarrollo, mayor extensibilidad. Por tanto, es vital dedicar tiempo para definirla atendiendo a las necesidades de nuestro sistema.

2.1.1. Decisiones principales de diseño

Las **decisiones principales de diseño** son todas aquellas decisiones importantes que afectan a los fundamentos de nuestro sistema. Tienen gran importancia porque con el

paso del tiempo, y con el avance del desarrollo, estas elecciones comienzan a asentarse, y se vuelven más difíciles de cambiar o rectificar. [4] Estas no solo se toman durante la concepción del sistema, si no que también pueden surgir durante el desarrollo y posterior evolución.

Por ejemplo, una decisión de diseño principal que se suele tomar en las fases tempranas del desarrollo es la elección de la topología para la solución. Optar entre desarrollar un servicio monolítico, o un sistema formado por microservicios, va a condicionar prácticamente todo el desarrollo.

Aun así, no todas las decisiones de diseño tienen por qué ser consideradas como principales. Hay detalles de implementación que pueden dejarse ambiguos a nivel de la arquitectura. Se concretarán durante en el momento en que se desarrolle.

Las decisiones principales de diseño normalmente se resumen en comparativas entre distintas alternativas, cada una de ellas con sus ventajas e inconvenientes. Pueden tomarse en base a distintos criterios. Entre ellos podemos destacar: [Citation needed]

- **Requisitos del sistema:** a partir del dominio y las necesidades de nuestros usuarios, podemos deducir: la funcionalidad a implementar, las restricciones que debemos respetar y otras propiedades que debe poseer el sistema.
- **Arquitectura actual:** las decisiones tomadas previamente también condicionan las elecciones que se tomen más adelante. Cuanto más avanza el desarrollo, más se asientan las decisiones previas, y más dificil es rectificarlas.
- **Experiencia previa:** del desarrollo de este u otros sistemas. Podemos obtener métricas del funcionamiento y uso de nuestro sistema para informar decisiones futuras. [Cita devops]

2.1.2. Componentes de una arquitectura

Otra posible definición de arquitectura la encontramos en el estándar IEEE 42010-2011 [7]: 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 usando tres conceptos: [5]

- **Elementos**: Son las piezas fundamentales que conforman el sistema. Implementan la 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 entre los elementos o el entorno de operación. Describe *cómo* está organizado el sistema. Por ejemplo: un servicio contacta con otro a través de una API.
- **Justificación**: Razonamiento o motivación de las decisiones que se han tomado. Responden al *por qué* algo se hace de determinada forma. Normalmente no pueden deducirse a partir de los elementos y la forma, por lo que es necesario describirlos.

Elementos

Durante el diseño del sistema, para lidiar con la complejidad que pudiera alcanzar, recurrimos a dos mecanismos: la abstracción y la separación de responsabilidades. [6] Aplicándolas repetidamente sobre los requisitos pueden sernos de gran ayuda para identificar distintas unidades de funcionalidad. Se trata de los componentes.

Según [4], 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: módulos dentro un mismo proceso, servicios distribuidos, etc.

Los componentes exponen una interfaz que permite acceso a esa funcionalidad o datos que encapsulan. A su vez, también declaran una serie de requisitos con otras interfaces de las que dependen para operar. En la figura 2.1 tenemos un ejemplo.

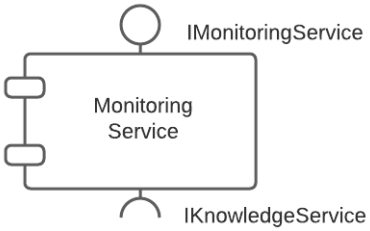

Figura 2.1: El servicio de monitorización representado como un componente. Ofrece una interfaz (*IMonitoringService*) y requiere de otra para funcionar (*IKnowledgeService*).

Por si solos, estos componentes independientes no aportan mucho valor. Son la unidad básica de composición: podemos conectar distintos componentes para que trabajen conjuntamente y realicen tareas más complejas. De esta forma podemos componer sistemas. Por ello, un aspecto clave es la integración y la interacción entre componentes. [8]

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. Representan la transferencia de datos y de control entre componentes. En la figura 2.2 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. De esta forma, podemos estudiar la arquitectura y elegir los mecanismos adecuados para cada interacción del sistema. [4].

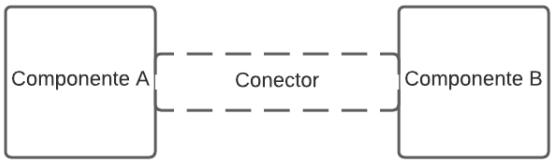

Figura 2.2: Ejemplo de comunicación de dos componentes a través de un conector.

A nivel de ¿diseño?, los conectores están compuestos por uno o más **conductos** o canales de comunicación. 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 [8]:

- **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 librerias. Por ejemplo: un balanceador de carga que redirige peticiones a los nodos.

Por tanto, una vez hemos decidido que dos componentes deben comunicarse, es momento de evaluar cuál es el mecanismo de comunicación más adecuado. Para ello, podemos consultar la taxonomía de conectores de [8]. Basándonos en nuestros requisitos, la arquitectura ya definida, y los mecanismos de despliegue que queremos usar, elegimos el conector más adecuado.

Forma

TODO

Justificación

Una vez definidos los componentes y los conectores, tendremos una representación del sistema. Pero se trata de una imagen incompleta. No cuenta con ciertos detalles que nos ayudan a entenderlo mejor, como las alternativas que se consideraron en el diseño y por qué se descartaron en favor de la elegida, entre otros. Tampoco cuenta con detalles minuciosos que pueden guiar a la implementación.

Es decir, necesitamos de un concepto adicional en nuestra arquitectura para describirlos. Se trata de la **justificación**. [5] Nos aporta detalles más precisos sobre la arquitectura.

2.1.3. Estilos arquitectónicos

Este conjunto de decisiones principales Una decisión de diseño principal que se suele tomar en las fases tempranas del desarrollo es la elección de un **estilo arquitectónico** para la solución. Se trata de una serie de decisiones de diseño prinEstos imponen una serie de restricciones en cuanto a la implementación del sistema, pero a su vez nos aporta una serie de beneficios... Optar por desarrollar un servicio monolítico o una arquitectura basada en microservicios va a condicionar prácticamente todo el desarrollo. Desde el diseño, la implementación, el testeo, y sobre todo, el despliegue y operación.

2.2 Arquitectura de la solución

Como comentamos en el Capítulo 1: Introducción, el objetivo del trabajo es adaptar un servicio monolítico para que funcione como un sistema distribuido basado en microservicios. Se trata de un cambio arquitectonico importante. Por ello, queremos diseñar una solución ingenieril teniendo en cuenta las particularidades del sistema.

Este servicio implementa un **bucle de control**, útil para dotar a un sistema con capacidades de computación autonoma. Especificamente, sigue la arquitectura del bucle MAPE-K[2, 3], que mostramos en la figura 2.3. El objetivo de partida fue separar cada uno de los componentes del bucle en microservicios independientes. AÑADIR IMAGEN EJEMPLO DEL BUCLE DESCOMPUESTO EN MICROSERVICIOS. BUSCAR REFERENCIAS DE LIBROS SOBRE DIVIDIR MONOLITOS.

Combinar con párrafo anterior: Actualmente, el sistema es un servicio monolitico que está muy acoplado a la solución. Queremos desacoplarlo para que pueda usarse la misma infraestructura para varios sistemas (*multi-tennancy*).

[1]Obtenido de: https://wwwvs.cs.hs-rm.de/vs-wiki/index.php/(WS12-01)_Cloud/Dokumentation

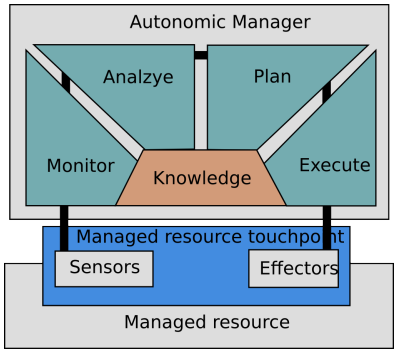

Figura 2.3: Arquitectura de un Bucle MAPE-K.[1]

2.2.1. Distribución de los componentes TODO: MUY INCOMPLETO

Por suerte, partimos de un sistema existente, con una arquitectura bien definida y documentada. Conociamos el rol de los componentes del servicio y sus requisitos. Asi 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?

Una vez determinadas las "fronteras.^entre los microservicios, hemos definido los componentes de nuestro sistema. Asi que, el primer problema al que nos enfrentamos estaba relacionado con la comunicación: si separamos las distintas etapas del bucle en microservicios, ¿cómo hacemos para que se comuniquen? Hay que tener en cuenta que estos pueden estar desplegados y replicados en distintas máquinas.

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

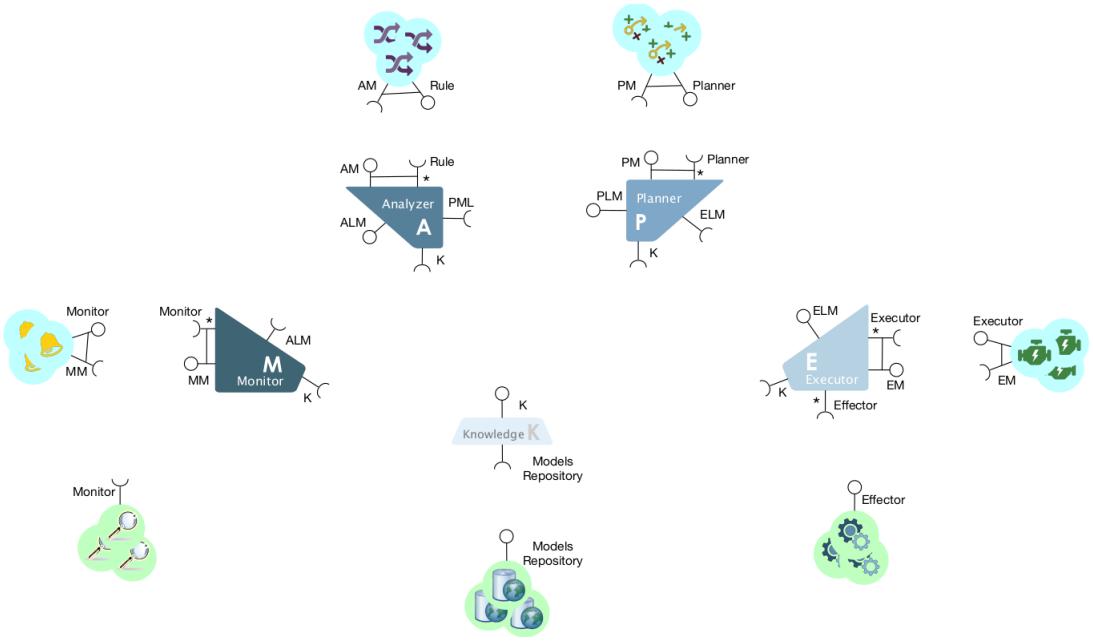

Figura 2.4: Diagrama con los componentes que forman nuestra arquitectura distribuida

En este caso, ponian como ejemplo un estilo arquitectónico que se llama C2 (components and connectors, creo) . La idea era tener capas o niveles de microservicios. Un microservicio en una capa concreta solo pueden contactar con sus vecinos (inmediata-

mente superior e inmediatamente inferior). Dentro del mismo nivel no pueden contactar entre ellos. Según la dirección de la comunicación, se utilizan mecanismos distintos:

- **Peticiones**: Un microservicio de más abajo en la jerarquía (o más externo), se comunica con un vecino que está más arriba. Por ejemplo, sería el caso de cuando el servicio de monitorización escribe una propiedad en el servicio de conocimiento.
- **Notificaciones**: Un microservicio de más arriba en la jerarquía (o mas interno), se comunica con un vecino que está por debajo de él. Por ejemplo, cuando el servicio de conocimiento notifica al servicio de análisis que ha cambiado una propiedad.

He encontrado una imagen que lo describe más o menos:

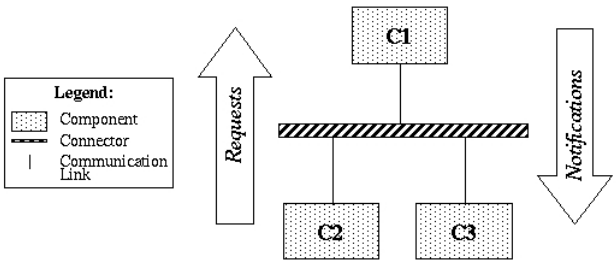

Figura 2.5: Ejemplo del estilo arquitectónico C2 (*Components and Connectors*). [1]

Mi propuesta entonces sería inspirarnos un poco en esta aproximación. Asi aprovechamos para dividir los microservicios en niveles y elegir los mecanismos para cada tipo de comunicación.

Yo distinguiría entre 4 niveles, desde más externo a más interno:

- **Más externo**: Sondas y Efectores. Son los elementos que ya tienen más contacto con el sistema gestionado por el bucle. Cuanto menos esté acoplado el bucle a ellos mejor.
- **Nivel de solución**: En esta capa se encuentran los microservicios específicos a una solución concreta. En el ejemplo que te mostré el otro día, sería el servicio de Room Monitor, que monitoriza la temperatura de una habitación
- **Nivel del bucle**: Aquí se encuentran los servicios del propio bucle: servicio de monitorización, análisis, planificación y ejecución. Esta capa no debería tener dependencias con los microservicios de la solución. Es justo lo queríamos abordar en este trabajo.
    - También se aprecia algo que implementé en mi solución: el servicio de monitorizacion hace de intermediario entre los monitores de la solución y el conocimiento. No se accede directamente a estas propiedades.
- **Conocimiento**: Es la capa más interna y la base de la arquitectura. Todos los microservicios de la capa del bucle dependen de ella para funcionar.

Me he basado en el diagrama típico de Clean Architecture para representar un poco la idea:

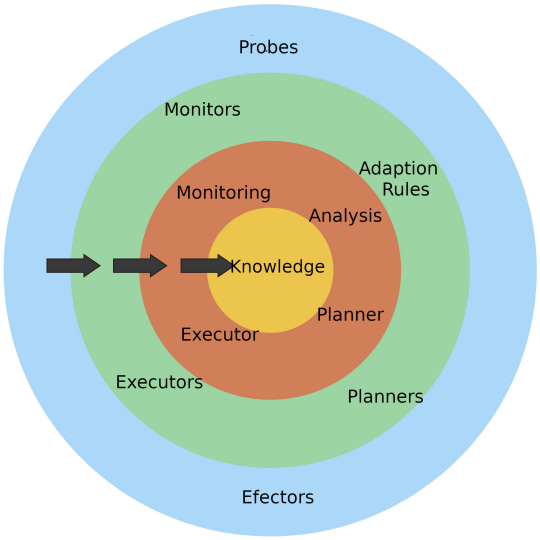

Figura 2.6: Representación de nuestra propuesta arquitectónica. Inspirado en Arquitectura Limpia (*Clean Architecture*). [2]

En el dibujo, las flechas de "fuera hacia adentrorepresentarían las Peticiones (o requests). Las notificaciones irían en sentido contrario.

Mi idea entonces es definir dos tipos de conectores:

- **Peticiones**: Utilizamos la estrategia que ya definimos en el hito anterior. Los servicios ofrecen APIs REST con una serie de operaciones. Las peticiones fluyen de microservicios más externos a los más internos, mediante llamadas HTTP.
  - Por ejemplo, en el hito anterior: Probe ->Monitor ->Monitoring Service ->Knowledge
  - Utilizamos OpenAPI para autogenerar clientes, así facilitamos la implementación de los clientes en cualquier lenguaje
- **Notificaciones**: En este caso optaría por usar brokers de mensajería (o algo parecido), de forma que los microservicios de capas más internos notifican a las más externas, sin necesidad de acoplarse directamente. Cada capa estaría suscrita a los eventos de la que esté por debajo de ella. Por ejemplo: Analysis Service se suscribe a Knowledge, las Reglas de Adaptación se suscriben al Analysis Service, etc.
  - Seguiría un poco la filosofía que comentamos hace un tiempo para independizar el módulo de análisis del de monitorización.
  - Podríamos investigar la idea que has comentado, de un conector que abstraiga al cliente de esta suscripción a una cola de mensajería. A esto no le he dado muchas vueltas aun.

2.2.2. Comunicación entre componentes

Una vez determinadas las ”fronteras” entre los microservicios, hemos definido los componentes de nuestro sistema. El siguiente problema al que nos enfrentamos fue la comunicación: si separamos las distintas etapas del bucle en microservicios, ¿cómo hacemos

[2] Imagen original de arquitectura limpia obtenida de: https://threedots.tech/post/ddd-cqrs-clean-architecture-combined/

para que se comuniquen? Debemos tener en cuenta que estos pueden estar desplegados en máquinas distintas.

Comenzamos entonces la búsqueda de los conectores más apropiados para cada par de componentes. Seguimos la estrategia descrita en [4] y consultando patrones de comunicación en sistemas distribuidos descritos en [9]. AMPLIAR Comenzamos eligiendo qué dos componentes queremos conectar.

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 consultar y actualizarlas durante su funcionamiento. En la figura 2.7 representamos inicialmente ambos componentes y un conector, sin especificar de qué tipo será.

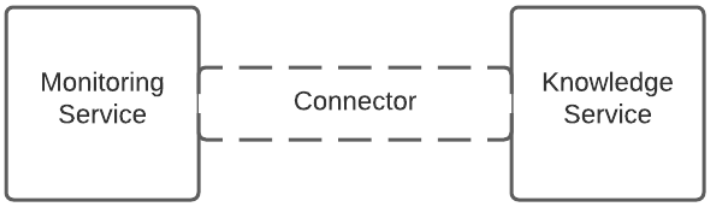

Figura 2.7: Boceto inicial: queremos conectar el servicio de monitorización con la base de conocimiento para poder leer propiedades de adaptación.

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.

Ahora, debemos identificar qué **tipos de conector** serían adecuados para nuestros componentes. 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.

Sabiendo esto, en lugar de recurrir a la taxonomía que lista [8], optamos por consultar las estrategias de comunicación habituales para sistemas distribuidos descritas en [9]. 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.

- **Invocación de métodos remotos** o (***Remote Procedure Call***): Esta aproximación se basa en el estilo cliente-servidor. En ella, 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 es capaz de 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. [10] En este caso, el programa cliente interactúa con objetos que se encuentran en servidores remotos. 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 devol-

verá un resultado. Así, abstraen al cliente de todo este proceso de comunicación. En la figura 2.8 tenemos un esquema de este mecanismo.

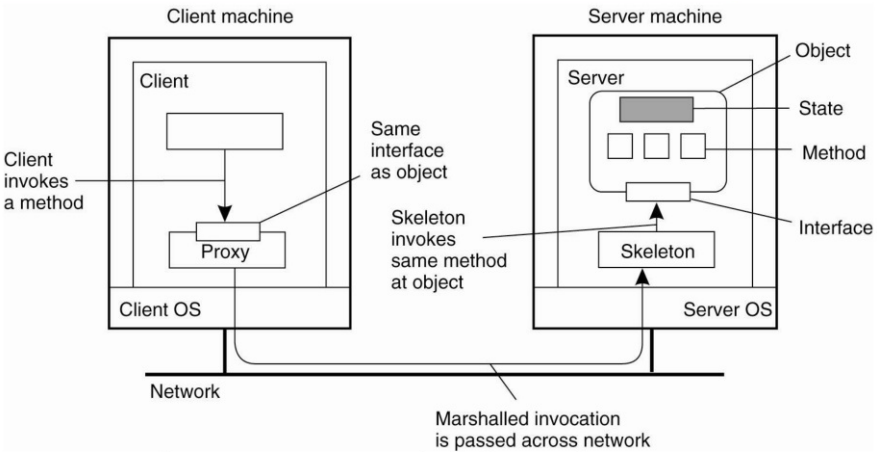

Figura 2.8: Funcionamiento del sistema de objetos distribuidos. [10]

- **Ventajas**:
  - Permite la distribución del procesamiento del sistema.
  - Abstrae al cliente de esta interacción con un servidor remoto. Para el cliente es 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.
- **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. [11]
  - Dificulta la integración. Cada servicio ofrece sus propias funciones distintas.
  - Si adoptamos sistemas como Java RMI, nuestro sistema se acopla a esa tecnología concreta. [9]. Nos resta 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. [4] Su concepto principal son los **recursos**: cualquier elemento del cual la API puede ofrecernos información y que pueda tener asociado un identificador único (una URI). [12] Por ejemplo, podrían ser las entidades del dominio que gestiona nuestro servicio: Usuarios, 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. [13] La implementación más común de REST es sobre el protocolo HTTP.

- **Ventajas**:
  - ***Stateless***: El servidor no mantiene el estado de la sesión. 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 la 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. [9]
  - **Generación de clientes**: Para facilitar la comunicación con APIs REST, podemos generar librerias cliente utilizando el estándar OpenAPI. Lo explicaremos con maś detalle en la sección 2.2.3.
- **Desventajas**:
  - **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 es difícil representar con los métodos ofrecidos por el protocolo de comunicación. Pueden requerir más tiempo de diseño, o incluso ser implementados siguiendo RPC.

- **GraphQL**[3] AMPLIAR: Se trata de un protocolo para que un cliente pueda hacer consultas personalizadas sobre los datos de un servidor. No necesitan que haya sido implementado con lógica asociada. De esta forma, se puede reducir la cantidad de peticiones a través de la red que se necesita ejecutar para obtener la misma información.

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

[3] Página oficial: https://graphql.org/

- ***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. [9] Hay varias estrategias de comunicación posibles: punto a punto, *publish-suscribe*, híbrida...

  Tomemos por ejemplo *publish-suscribe*: es una estrategia 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 subscritos a este tema en cuanto sea posible. [14] En la figura 2.9 tenemos un ejemplo de esta estrategia.

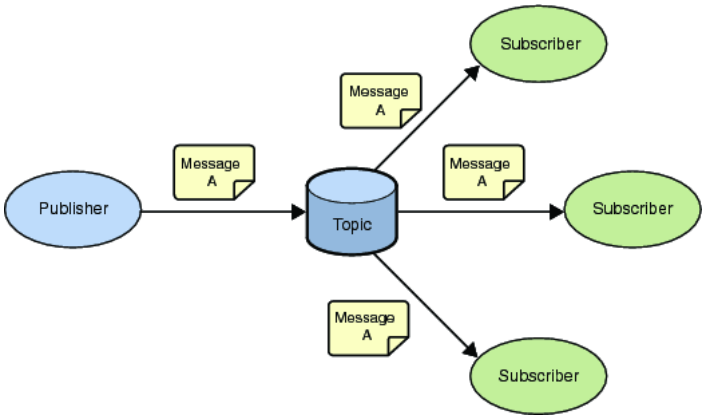

Figura 2.9: Estrategia *publish/suscribe*: el *broker* actúa como intermediario en la comunicación *multicast*. Imagen obtenida de [4]

La mayor ventaja de este estilo de comunicación es el **desacoplamiento** entre los servicios. [15] 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 la dirección del *broker* para enviar o recibir mensajes.

- **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.
  - **Desacoplamiento de los servicios**: Ni los productores ni los consumidores necesitan conocer el origen o destino de sus mensajes.
  - **Envío garantizado de mensajes**: El *broker* garantiza que el mensaje será entregado *al menos* una vez.
- **Desventajas**:
  - **Requisitos de infraestructura**: Utilizar un *broker* de mensajería implica tener ciertas consideraciones a la hora de diseñar nuestro plan de despliegue. Estos sistemas requieren de replicación para alcanzar la alta disponibilidad necesaria para que sea fiable. [9] Por tanto, aumenta la complejidad de operar el sistema.
  - **Envío garantizado de mensajes**: Para poder garantizar el envío de un mensaje, el *broker* puede recurrir a reenviarlo. Debemos diseñar nuestros

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

sistemas de forma que estos mensajes duplicados, si ya han sido procesados, deben ser descartados.

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. Siguiendo también este razonamiento nos llevó a descartar el *broker* de mensajería. 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 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. Implementamos ambas funciones mediante *endpoints* HTTP. Su especificación se detalla a continuación en las tablas 2.1 y 2.2.

<table>
<tr><td>Operación HTTP</td><td colspan="2">GET</td><td>Ruta</td><td>property/{propertyName}</td></tr>
<tr><td>Descripción</td><td colspan="4">Devuelve el valor de la propiedad, si existe.</td></tr>
<tr><td>Parámetros</td><td colspan="2">propertyName</td><td colspan="2">El nombre de la propiedad que deseamos obtener. Se lee a partir de la ruta de la petición.</td></tr>
<tr><td rowspan="3">Respuestas posibles</td><td colspan="2">Código 200 (Ok)</td><td colspan="2">La propiedad se ha encontrado. Incluye un payload con el siguiente esquema:
Value: Valor de la propiedad serializado en JSON.
LastModification: Fecha y hora de la última modificación de esta propiedad.</td></tr>
<tr><td colspan="2">Código 400 (Bad request)</td><td colspan="2">La petición está mal formada, no es acuerdo al contrato.</td></tr>
<tr><td colspan="2">Código 404 (Not found)</td><td colspan="2">No se ha encontrado ninguna propiedad con el nombre proporcionado.</td></tr>
<tr><td>Ejemplo</td><td colspan="4">Petición para obtener la propiedad currentTemperature:
<pre>
Request:
HTTP GET property/currentTemperature

Response: 200 Ok
{
  value: {
    "Value":16.79,
    "Unit": 1, // Celsius
    "ProbeId":"c02234d3-329c-4b4d-aee0-d220dc25276b",
    "DateTime":"2022-01-15T18:19:38.5231231Z"
  },
  lastModification: "2022-01-15T18:19:39.123213Z"
}
</pre></td></tr>
</table>

Tabla 2.1: Especificación de la operación para obtener una propiedad del servicio de conocimiento.

<table>
<tr><td>Operación HTTP</td><td>PUT</td><td>Ruta</td><td>property/{propertyName}</td></tr>
<tr><td>Descripción</td><td colspan="3">Actualiza (o crea, si no existe) el valor de la propiedad con el nombre dado.</td></tr>
<tr><td rowspan="2">Parámetros</td><td>propertyName</td><td colspan="2">El nombre de la propiedad que deseamos crear o actualizar. Se lee a partir de la ruta de la petición.</td></tr>
<tr><td>SetPropertyDTO</td><td colspan="2">Un DTO que contiene el valor a asignar en la propiedad serializado en JSON. El DTO se encuentra en el cuerpo de la petición.</td></tr>
<tr><td rowspan="2">Respuestas posibles</td><td>Código 204 (No content)</td><td colspan="2">La propiedad se ha creado o actualizado correctamente. No incluye payload en el cuerpo de la respuesta.</td></tr>
<tr><td>Código 400 (Bad request)</td><td colspan="2">La petición está mal formada, no es acuerdo al contrato.</td></tr>
<tr><td>Ejemplo</td><td colspan="3">Petición para actualizar la propiedad currentTemperature con una medición de un termómetro:
Request:
HTTP PUT property/currentTemperature

{
  value: {
    "Value":16.79,
    "Unit": 1, // Celsius
    "ProbeId":"c02234d3-329c-4b4d-aee0-d220dc25276b",
    "DateTime":"2022-01-15T18:19:38.5231231Z"
  }
}

Response: 204 (No content)</td></tr>
</table>

Tabla 2.2: Especificación de la operación para actualizar o crear 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.

2.2.3. Open API

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. [16] Más adelante mostraremos un ejemplo, con el listing 2.2.

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. [17]

Uno de los casos de uso más interesantes es la generación de código a partir de la definición. Existen una serie de generadores[5] 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 2.1.

En el fragmento 2.1, 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).

[5] https://github.com/OpenAPITools/openapi-generator

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
       property. </response>
7  /// <response code="404"> The property was not found. </response>
8  /// <response code="400"> There was an error with the provided arguments. </
       response>
9  [HttpGet("{propertyName}")]
10 [ProducesResponseType(typeof(PropertyDTO), StatusCodes.Status200OK)]
11 [ProducesResponseType(StatusCodes.Status404NotFound)]
12 [ProducesResponseType(StatusCodes.Status400BadRequest)]
13 public IActionResult GetProperty([FromRoute]string propertyName)
14 {
15     if (string.IsNullOrEmpty(propertyName))
16     {
17         return BadRequest();
18     }
19
20     bool foundProperty = properties.TryGetValue(propertyName, out PropertyDTO
           property);
21
22     if (!foundProperty)
23     {
24         return NotFound();
25     }
26
27     return Ok(property);
28 }
```

Listing 2.1: 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 2.2, 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
                  property.",
22            "content": {
23              "application/json": {
24                "schema": {
25                  "$ref": "#/components/schemas/PropertyDTO"
26                }
27              }
28            }
29          },
30          "404": {
31            "description": "The property was not found.",
32          },
33          "400": {
34            "description": "There was an error with the provided arguments.",
35          }
36        }
37      }
38    }
```

Listing 2.2: 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 2.1. Encontramos grandes similitudes con la especificación presentada en la tabla 2.1.

Finalmente, la arquitectura del conector que emplearemos para implementar las peticiones aparece en la figura 2.10. 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*.

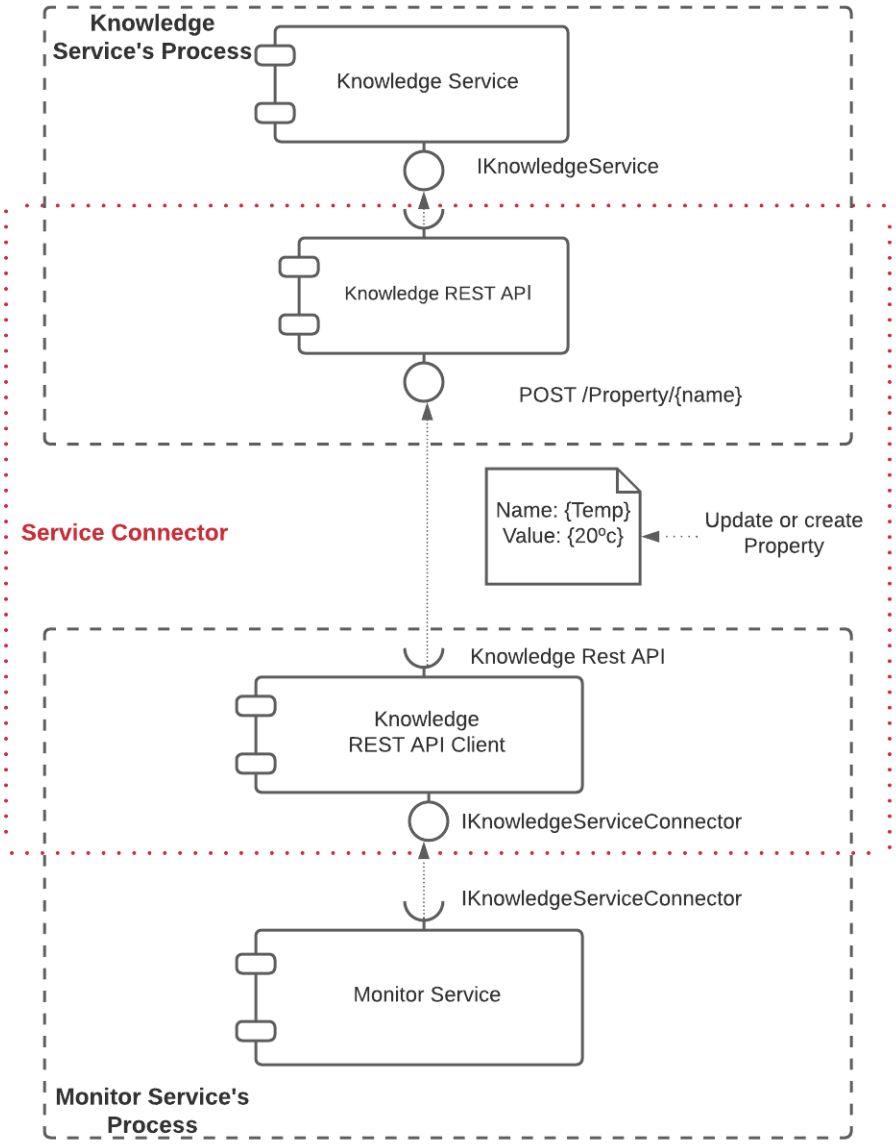

Figura 2.10: Diseño del conector usando implementación Cliente - Servidor

CAPÍTULO 3
Conclusions

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

Bibliografía

[1] "UCI Software Architecture Research - UCI Software Architecture Research: C2 Style Rules."

[2] "An Architectural Blueprint for Autonomic Computing," tech. rep., IBM, 2006.

[3] 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.

[4] R. N. Taylor, N. Medvidovic, and E. Dashofy, *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, Jan. 2009.

[5] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, Oct. 1992.

[6] R. C. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series, London, England: Prentice Hall, 2018.

[7] IEEE, ISO, and IEC, "Standard 42010-2011 - Systems and software engineering – Architecture description," tech. rep., 2011.

[8] 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.

[9] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Aug. 2021.

[10] 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.

[11] P. Jausovec, "Fallacies of distributed systems," Nov. 2020.

[12] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, May 2007.

[13] M. Nally, "REST vs. RPC: What problems are you trying to solve with your APIs?," Oct. 2018.

[14] RabbitMQ, "Publish/Subscribe documentation."

[15] J. Korab, *Understanding Message Brokers*. O'Reilly Media, June 2017.

[16] OpenAPI, "OpenAPI Specification v3.1.0."

[17] D. Westerveld, "Chapter 3: OpenAPI and API Specifications," in *API Testing and Development with Postman*, Packt Publishing, May 2021.

APÉNDICE A

Configuració del sistema

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

A.1 Fase d'inicialització

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

A.2 Identificació de dispositius

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

APÉNDICE B
??? ????????????? ????

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