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 [1, 2], 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.

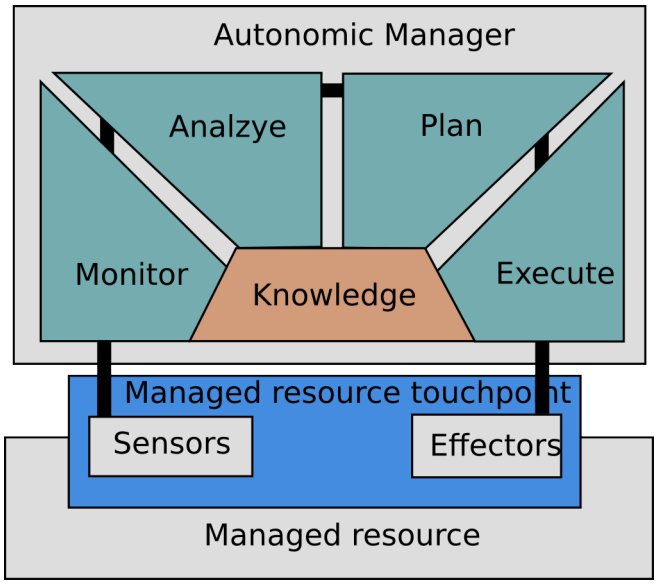

Figura 1.1: Representación del Bucle MAPE-K.[1]

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: [1]

- **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
Arquitecturas de *software*

Según [3], la **arquitectura de un sistema *software*** es el conjunto de todas las decisiones de diseño principales que se toman durante la vida del sistema, aquellas que sientan las bases del desarrollo. Se podría establecer un simil con los planos de construcción de un edificio. Estas decisiones no solo se toman durante su concepción, si no también durante su desarrollo y posterior evolución.

La arquitectura afecta a todos los apartados del sistema: su estructura, funcionalidad, la implementación...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 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. Por tanto, es vital dedicar tiempo para definir la arquitectura en base a las necesidades de nuestro sistema.

2.1 Decisiones principales de diseño

Las decisiones principales de diseño normalmente se resumen en comparativas entre distintas alternativas, cada una de ellas con sus ventajas e inconvenientes. Con el paso del tiempo, y con el avance del desarrollo, estas elecciones comienzan a asentarse, y se vuelven más dificiles de cambiar o rectificar.

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]

2.2 Componentes de una arquitectura

Según el estándar IEEE 42010-2011 [4], la arquitectura de un sistema es "*un conjunto de conceptos o propiedades fundamentales, personificados por sus elementos, sus relaciones, y los principios que guían su diseño y evolución*". Nos provee de un marco de referencia común, de decisiones que nos nos provee con un vocabulario común, que nos permite describir sistemas

Por tanto, podemos describirla usando tres conceptos: [5]

- **Componentes:** Son las piezas fundamentales que componen el sistema. Implementan la funcionalidad de la aplicación. Se utilizan para describir *qué* partes conforman 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.

La arquitectura de un sistema puede contar con diferentes vistas, según aquel aspecto que deseemos resaltar. Por ejemplo, puede interesarnos más la interacción entre los componentes. O cosas por el estilo.

Durante el diseño, para lidiar con la complejidad que pudiera alcanzar el sistema, solemos recurrir a descomponerlos usando diseños modulares: sistemas compuestos por unidades de funcionalidad que tienen una función específica. [3] Estos elementos funcionales son los componentes. Dependiendo de las características de nuestro sistema, pueden tomar distintas formas: módulos dentro un mismo proceso, servicios distribuidos, etc.

Como hemos comentado antes, un sistema está conformado por **componentes** que implementan la funcionalidad de la aplicación. No suelen trabajar de forma aislada, si no que trabajan conjuntamente para realizar tareas más complejas. Por tanto, un aspecto clave es la integración y la interacción entre ellos. [6]

Para que dos o más componentes puedan interactuar, necesitamos definir un mecanismo de comunicación. Para ello, recurrimos a los **conectores**: se trata de elementos arquitectónicos que nos ayudan a diseñar y razonar sobre la comunicación entre componentes. Representan la transferencia de datos y de control entre componentes. En la figura 2.1 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. [3].

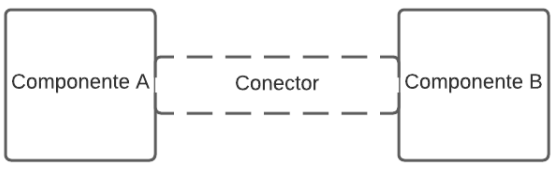

Figura 2.1: Ejemplo de comunicación de dos componentes a través de un conector.

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 [6]:

- **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 [6]. Basándonos en nuestros requisitos, la arquitectura ya definida, y los mecanismos de despliegue que queremos usar, elegimos el conector más adecuado.

2.3 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[1, 2], que mostramos en la figura 2.2. 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.

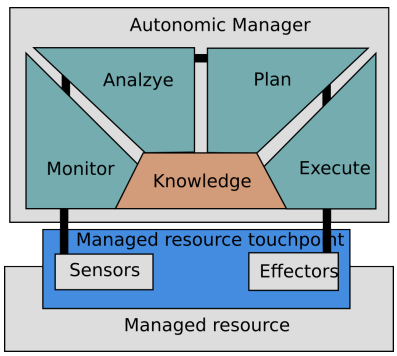

Figura 2.2: Arquitectura de un Bucle MAPE-K.[1]

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

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*).

2.3.1. Distribución de los componentes

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

TODO: Layered architecture. Dos capas: Distinguimos entre los microservicios clientes (monitores / sensores, reglas especifícas, etc) (círculo externo) y los microservicios de nuestro bucle MAPE-K (monitoring service, knowledge service...) (círculo interno). De esta forma podemos distinguir las responsabilidades de cada capa y como ocurre la comunicación entre ellas. Restringir la dirección de la comunicación por niveles. Comandos hacia arriba, notificaciones hacia abajo. Por ej: Podría ocurrir que cuando la información viene del dominio "hacia dentro", la comunicación sea a través de APIs REST y clientes autogenerados con OpenAPI. Pero, cuando la información fluye del bucle "hacia fuera", por ej del Analyser a las reglas, se hace mediante brokers de mensajería ->Menos acoplamiento.

AMPLIAR

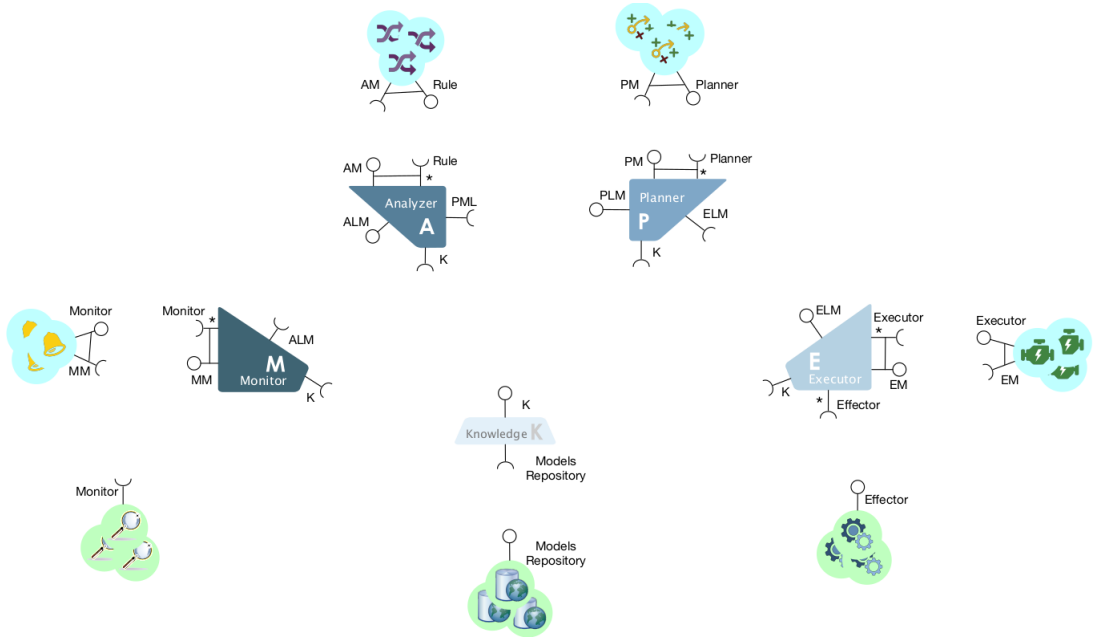

Figura 2.3: Diagrama con los componentes que forman nuestra arquitectura distribuida

2.3.2. 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 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 [3] y consultando patrones de comunicación en sistemas distribuidos descritos en [7]. 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 2.4 representamos inicialmente ambos componentes y un conector, sin especificar de qué tipo será.

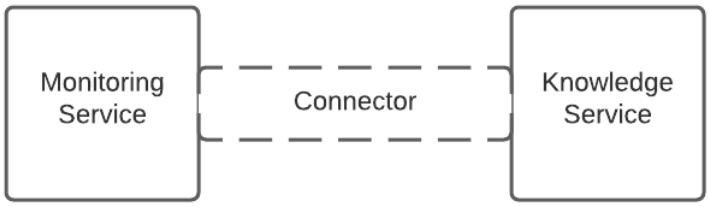

Figura 2.4: Boceto inicial: queremos conectar el servicio de monitorización con la base de conocimiento para poder leer propiedades de adaptación.

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 [6], optamos por consultar las estrategias de comunicación habituales para sistemas distribuidos descritas en [7]. 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***): Se trata de un paradigma de objetos distribuidos. [3] Nuestro programa interactúa con objetos que se encuentran en servidores remotos. Suele implementarse con objetos que actúan como un *proxy*: ofrecen una interfaz para que el cliente invoque sus funciones localmente. Estas funciones internamente realizan una llamada al servicio remoto, donde se encuentre este objeto. El servidor remoto procesa la petición y nos devolverá un resultado. Así, abstraen al cliente de todo este proceso de comunicación.

Estos *proxies* suelen generarse a partir de un contrato, que define que operaciones ofrecen estos objetos. Existen varios protocolos que implementan este mecanismo, como Java RMI, gRPC o SOAP.

- **Ventajas**:
  - Permite la distribución del procesamiento del sistema.
  - Abstrae al cliente de esta interacción a un servidor remoto. No tenemos que implementar la interacción con el servidor.
  - Al haber un contrato definido, podemos generar los *proxies* o clientes basados en este contrato. Por ejemplo: SOAP con WDSL o gRPC.
- **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. [8]
  - Si adoptamos sistemas como Java RMI, nuestro sistema se acopla a esa tecnología concreta. [7]. 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 trata de un estilo arquitectónico basado en el estilo cliente - servidor, pero con ciertas restricciones adicionales. [3] Su concepto principal son los **recursos**: cualquier elemento que pueda tener asociado un identificador (una URI). [9] Sobre los recursos podemos ejecutar una serie de acciones definidas por el protocolo de comunicación. Normalmente, el protocolo es HTTP.

  - **Ventajas**:
    - **Escalable**:
    - ***Stateless***: El servidor no mantiene el estado de la operación.
    - **Interoperabilidad**: Ampliamente utilizado en servicios de Internet. Es ideal para que clientes externos contacten con nuestro sistema mediante peticiones síncronas. [7]
  - **Desventajas**:
    - **Rendimiento**: El rendimiento es peor comparado con cualquier mecanismo RPC binario. La información serializada en XML o JSON es mayor que si estuviera en un formato binario.
    - **Rendimiento**: El rendimiento es peor comparado con cualquier mecanismo RPC binario. La información serializada en XML o JSON es mayor que si estuviera en un formato binario.

- **GraphQL**[2]: 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.

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

- **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.

- ***Brokers* de mensajería**:

De estas cuatro opciones, podemos descartar inmediatamente la opción de GraphQL. Se trata de un conector más orientado a las consultas de datos. Nosotros necesitamos ejecutar escrituras de los valores de las propiedades. Aunque podría implementarse para el servicio de las lecturas, no preveemos que las consultas de las propiedades requieran de agregación de datos. Simplemente se obtiene el valor de una propiedad.

Por otro lado, una de nuestras prioridades es la interoperabilidad, para que cualquier microservicio que implemente esta API peuda comunicarse con nuestro servicio. Finalmente, queríamos que la comunicación fuese síncrona a la hora de solicitar propiedades, por lo que nos terminamos decantando por REST sobre HTTP.

Aun así, queríamos aprovechar algunas de las ventajas que ofrece RPC para simplificar la integración de nuestra API con las librerias clientes.

†A continuación, debemos identificar qué **rol** debe jugar este conector. Se trata de cubrir la **comunicación** entre dos componentes: necesitamos enviar y recibir datos del servicio de persistencia. También detectamos que es necesaria la **coordinación** entre los componentes. Por ejemplo, para la lectura de propiedades debemos contactar con otro servicio, y esperar su respuesta. Hay por tanto una transferencia de control entre ellos. Respecto al resto de roles, de momento no hemos detectado necesidad de conversión o facilitación.

En este caso, hemos optado por una mezcla de RPC implementado sobre HTTP. No se trata de una implementación del todo rest. Pero de esta forma, podíamos evitar acoplarnos a una tecnología concreta, como describe el libro. Nos parecía más razonable que el monitor exponga directamente una API HTTP pública. De esta forma, podríamos explotar el uso de OpenAPI, un DSL para describir APIs que explicaremos a continuación.

Así, mejoramos la compatibilidad con cualquier cliente, y nos daba más flexibilidad.

Ya tenemos la parte de la API, expuesta por el servidor. Ahora nos queda la parte del cliente. ¿Cómo contactamos desde el monitor con la API? Implementamos las llamadas directamente con un cliente HTTP? Esto será muy costoso de mantener, y no será resiliente a los cambios. Será muy complicado.

Por suerte, a partir de la especificación OpenAPI, podemos generar un componente que haga de cliente de la solución. Este cliente hace de Proxy y nos abstraen de la lógica para establecer la conexión, el protocolo de comunicación, formato de los mensajes, etc.

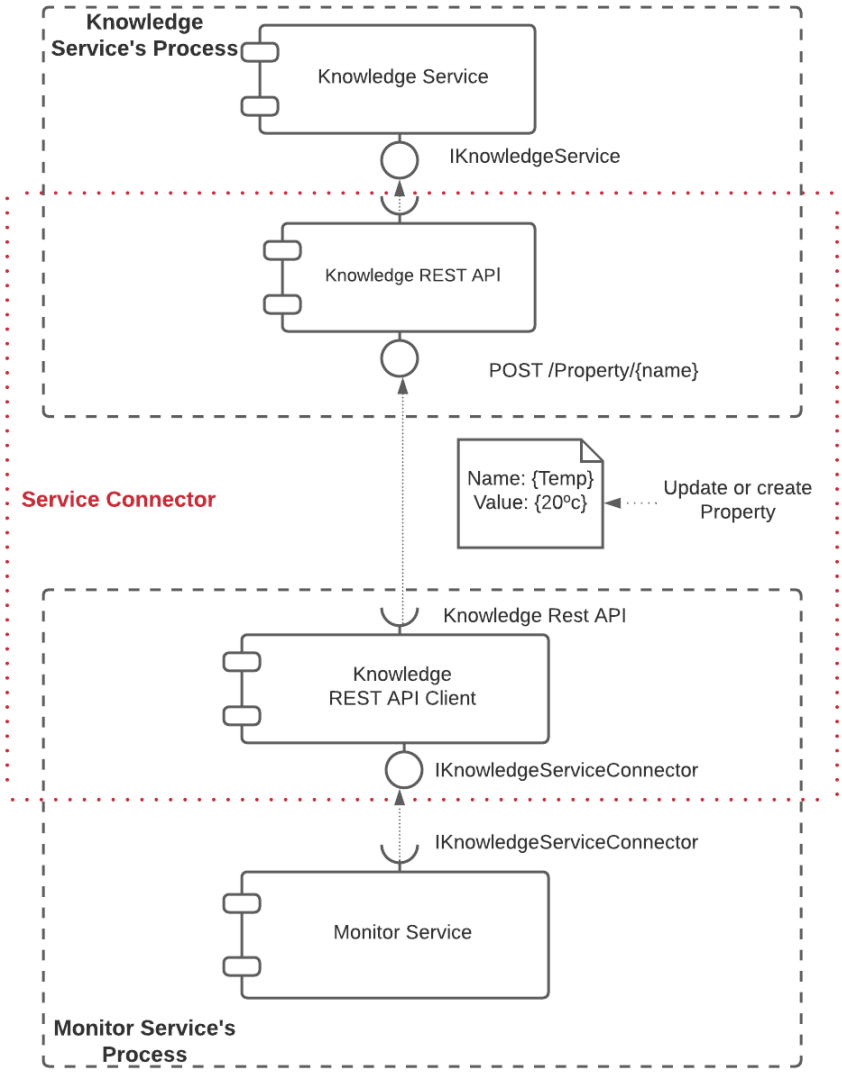

Figura 2.5: Diseño del conector usando implementación Cliente - Servidor

2.3.3. Open API

OpenAPI[3] es un lenguaje estándar para describir APIs implementadas sobre el protocolo HTTP. Es un *domain specific language* que podemos utilizar para describir APIs de nuestras aplicaciones

Para el desarrollo de este trabajo, hemos optado por utilizar conectores basados en APIs REST. Nos permitía utilizar mecanismos ya presentes en los microservicios que queríamos desarrollar, y que fueran más homogéneos. Para facilitar la compatibilidad de los microservicios, y facilitar el desarrollo de nuevos microservicios, hemos decidido usar la especificación OpenAPI.

La principal ventaja que nos ofrecía OpenAPI era la posibilidad de generar código a partir de la especificación. Permite generar código tanto del cliente como del servidor. De esta forma, podíamos implementar nuestra API genérica en un lenguaje . En este caso,

[3]Open API specification: https://spec.openapis.org/oas/latest.html

se decidió utilizar conjuntamente el lenguaje C# junto con el framework ASP.NET Core, para implementar los microservicios iniciales.

A partir de estos microservicios, podemos exportar la especificación de OpenAPI, haciendo uso de los endpoints, atributos y comentarios. Por ejemplo, en este endpoint del servicio de monitorización, vemos cómo obtiene una propiedad del servicio de conocimiento.

Podemos observar cómo el método está decorado con atributos que describen el tipo de respuesta que produce, según el código de respuesta HTTP. Estos comentarios se utilizan en la generación de la especificación para obtener mejor implementación

Después, haciendo uso de las librerías de generación de código de OpenAPI.

CAPÍTULO 3
Conclusions

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

Bibliografía

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

[2] 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.

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

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

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

[6] 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.

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

[8] P. Jausovec, "Fallacies of distributed systems." https://blogs.oracle.com/developers/post/fallacies-of-distributed-systems, Nov. 2020.

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

APÉNDICE A

Configuració del sistema

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

A.1 Fase d'inicialització

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

A.2 Identificació de dispositius

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

APÉNDICE B
??? ????????????? ????

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