UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
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

La Computación Autónoma (*Autonomic Computing*) promueve la ingeniería, diseño y desarrollo de sistemas con capacidades de auto-adaptación, a través del uso de bucles de control. Estas capacidades le confieren a estos sistemas la posibilidad de adaptarse a entornos cambiantes, a conflictos operacionales e incluso a la optimización dinámica en su ejecución. Por otra parte, en la última década, la computación en el cloud y las arquitecturas basadas en microservicios se han postulado como una infraestructura muy flexible y dinámica para desplegar soluciones altamente disponibles y eficientes. Hay una tendencia clara a aplicar este tipo de infraestructuras, gracias a los múltiples beneficios que aporta.

En este trabajo se explorará cómo diseñar soluciones que, aplicando los conceptos de los bucles de control (AC), estén preparadas para desplegarse en la nube. Para ello se tomará como punto de partida la infraestructura FaDA (desarrollada por el grupo PROS/Tatami del instituto VRAIN/UPV) que propone una estrategia para realizar la ingeniería de sistemas auto-adaptativos usando bucles de control MAPE-K.

Como resultado de este TFM se espera obtener la definición arquitectónica de soluciones auto-adaptativas (incluyendo tanto al bucle de control MAPE-K como directrices para la implementación de los diferentes componentes adaptativos de la solución) diseñadas para desplegarse nativamente como microservicios en la nube. Por último, se aplicará la propuesta realizada al desarrollo de un caso práctico para demostrar su viabilidad y aplicabilidad.

Palabras clave: computación autónoma, arquitecturas de microservicios, bucles de control, MAPE-K, computación en la nube

Abstract

????

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

Índice general

Apéndice

Índice de figuras

Índice de tablas

Índice de algoritmos

CAPÍTULO 1
Introducción

La revolución digital[1] ha permeado en todos los aspectos de nuestras vidas. En nuestro día a día usamos una gran variedad de aplicaciones informáticas: redes sociales, ofimática, comercios electrónicos...Muchas de ellas se encuentran alojadas en la red, en servidores externos.

Para las aplicaciones web, uno de sus requisitos claves es la **disponibilidad**. [1] Nuestros servicios deben estar en funcionamiento en todo momento para atender a nuestros usuarios. Tomemos por ejemplo el caso de una tienda *on-line*. La plataforma debe estar disponible el mayor tiempo posible. Si surgiera una incidencia y se degrada la capacidad de atender a clientes, o directamente no podemos atender a ninguno, perderemos ingresos.

Para atender estas incidencias, no es efectivo depender de operarios humanos. [2] Es muy costoso tener a alguien pendiente de la aplicación las veinticuatro horas del día para solucionar las incidencias. Debido a esto querremos que nuestro sistema sea capaz de **adaptarse automáticamente** a las distintas situaciones que surjan durante su operación. Recurrir al operario humano debería ser el último recurso.

En el ámbito de la computación autónoma (*autonomic computing*) encontramos el concepto de **sistemas auto-adaptativos**. Son sistemas capaces de ajustar su comportamiento en tiempo de ejecución en base su estado y el del entorno para alcanzar sus objetivos de operación. [2] Esto es posible mediante el uso de **bucles de control**. [3] Gracias a ellos, podremos dotar a los sistemas de capacidades para adaptarse a entornos cambiantes, resolver conflictos operacionales e incluso a la optimizarse dinámicamente.

Siguiendo con el ejemplo de la tienda on-line, un ejemplo de auto-adaptación sería adaptarse a los picos de demanda. Cuando tengamos mayor afluencia de clientes, debe ser capaz de aumentar su capacidad de cómputo. En cambio, cuando la afluencia baje, deberá ser capaz de reducirla.

1.1 Motivación

En este trabajo se quiere explorar el diseño de soluciones auto-adaptativas que estén preparadas para desplegarse en la nube. Para ello se tomó como punto de partida la infraestructura FaDA[2] (desarrollada por el grupo PROS/Tatami[3] del instituto VRAI-

[1] https://es.wikipedia.org/wiki/Revoluci%C3%B3n_Digital
[2] Página oficial: http://fada.tatami.webs.upv.es/
[3] Página oficial: http://www.pros.webs.upv.es/

N/UPV[4]). Esta propone una estrategia para la ingeniería de sistemas auto-adaptativos usando bucles de control MAPE-K[2, 4].

Actualmente, el bucle de control de FaDA está implementado como un servicio monolítico. Todos sus componentes operan dentro del mismo proceso, incluidos los específicos para sistemas manejados (sondas, monitores...). Se trata por tanto de una implementación muy rígida. En caso de querer modificar algún componente, hay que redesplegarlo entero.

Por ello, se buscó **dividir su funcionalidad en microservicios**. Es decir, cambiar la topología de la solución. Con ello, lograríamos independizar los componentes y su despliegue. Además, facilitaría escalar horizontalmente la capacidad del sistema en base a la carga.

1.2 Objetivos

Para el desarrollo del trabajo nos planteamos los siguientes objetivos:

1. Diseñar una arquitectura para soluciones auto-adaptativas preparadas para desplegarse nativamente como microservicios en la nube. Esto implica determinar los componentes en los que dividiremos la funcionalidad del bucle y los mecanismos de comunicación para conectarlos.
2. Definir directrices para la implementación de los diferentes componentes adaptativos específicos de una solución: monitores, sondas, efectores...
3. Desarrollar un caso práctico para demostrar la viabilidad y aplicabilidad de nuestra propuesta.

1.3 Estructura de la memoria

El trabajo se puede dividir en tres grandes secciones. La primera de ellas es el **marco teórico**. En el capítulo 2 hacemos una introducción a algunos conceptos de la computación autónoma y los bucles de control. Presentaremos la arquitectura MAPE-K, en la que se basa el trabajo. También describiremos algunos conceptos de arquitecturas de *software* que nos serán de interés.

La segunda parte de este trabajo trata sobre la **migración del sistema existente** a una arquitectura basada en microservicios. Para ello, comenzaremos describiendo la arquitectura del sistema actual en el capítulo 3. En base a este, en el capítulo 4 describiremos nuestra propia propuesta arquitectónica. Aquí se describirá los distintos componentes que conforman nuestra solución y se describirá los mecanismos de comunicación por los que optamos. Finalmente, en el capítulo 5 describimos nuestra implementación de referencia.

La última sección del trabajo es la referente al **caso de estudio** (capítulo 6). En él implementamos un sistema auto-adaptativo básico para un sistema de climatización. Nos sirvió para aplicar nuestra arquitectura y verificar su funcionamiento.

Cerramos el trabajo presentando las conclusiones (capítulo 7). En este apartado describiremos también las vertientes por las que se podría continuar ampliando el trabajo en un futuro.

Añadir diagrama de gant con los 7 hitos

[4]Página oficial: https://vrain.upv.es/

CAPÍTULO 2
Contexto Tecnológico

En este capítulo presentamos algunos de los conceptos más relevantes para el trabajo. Entre ellos se incluyen las arquitecturas de *software*, la computación autónoma y los bucles de control. Estos conceptos nos acompañarán a lo largo de la memoria.

2.1 Arquitecturas de *Software*

En esta sección haremos una breve introducción a las arquitecturas de *software*. Describiremos su motivación y los elementos que las componen. Esta sección es interesante por dos motivos:

- En el trabajo tratamos la migración de un sistema con arquitectura monolítica a una distribuida basada en microservicios. Trabajamos con componentes, conectores y otros elementos arquitectónicos.
- Por otro lado, el bucle MAPE-K es capaz de cambiar la arquitectura del recurso manejado en tiempo de ejecución. Sus adaptaciones se describen en base a operadores arquitectónicos: añadir o eliminar componentes, conectar o desconectarlos...

2.1.1. Definición

Según [5], 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. [5] 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 y desarrollar nuevas funcionalidades. Se convierte en una "gran bola de barro". [6]

Para evitarlo, es vital dedicar tiempo para plantear y definir una buena arquitectura. Según [7], una buena arquitectura es aquella que es «*fácil de entender, fácil de desarrollar, fácil de mantener y fácil de desplegar*». Esto se traducirá en una reducción de costes de mantenimiento y operación.

2.1.2. Componentes de una arquitectura

Otra posible definición de arquitectura la encontramos en el estándar IEEE 42010-2011 [8]: 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 estos tres conceptos: [9]

- **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, un conector...
- **Forma**: El conjunto de propiedades y relaciones de un elemenento 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. Nos aporta detalles más precisos sobre el sistema que no se pueden representar mediante los elementos o la forma. Un ejemplo podría ser qué alternativas se consideraron para tomar una decisión; y por qué se descartaron en favor de la elegida.

Para este trabajo, nos interesan especialmente los elementos. Concretamente los componentes y los conectores.

Componentes

El primer tipo de elemento que debemos tratar son los componentes. Según [5], 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 2.1 tenemos un ejemplo. *Monitoring Service* expone la interfaz *IMonitoringService*. Para poder funcionar, depende de un componente que ofrezca *IKnowledgeService*.

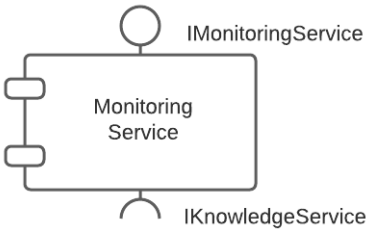


Figura 2.1: El servicio de monitorización representado como un componente. Ofrece una interfaz (*IMonitoringService*) y depende de otra para funcionar (*IKnowledgeService*).

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**. [10] La integración y la interacción entre ellos son aspectos clave que debemos abordar.

Conectores

Para que los componentes puedan interactuar, necesitamos definir uno o más mecanismos de comunicación. Recurriremos entonces a los **conectores**. Se trata de elementos

arquitectónicos que nos ayudan a investigar y especificar la comunicación entre componentes. [9] Son elementos independientes a la aplicación. No están acoplados a componentes específicos. Son por tanto **reutilizables**. [11]

Internamente, están compuestos por uno o más **conductos** o canales. A través de estos se realiza la transmisión de información. Según su **cardinalidad**, estos podrán conectar más o menos componentes. Hay una gran variedad de conductos disponibles: comunicación interproceso, en red, etc. Clasificamos los conectores según la complejidad de los conductos que utilizan [10]:

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

Una vez 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 [10].

Fijémonos por ejemplo en la figura 2.2. En ella mostramos dos elementos que queremos comunicar. Vemos que no se ha especificado todavía ningún detalle sobre cómo se implementará. Esto nos permitirá estudiar sus necesidades y elegir el mecanismo óptimo para la interacción. [5].

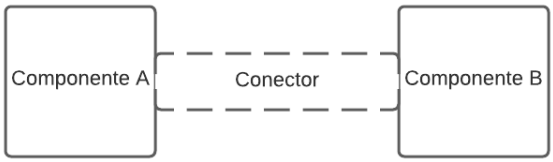


Figura 2.2: Ejemplo de comunicación de dos componentes a través de un conector.

2.2 Computación autónoma y bucles de control

Según [2], la **computación autónoma** tiene como objetivo dotar a los sistemas de **autonomía** en su operación. Es decir, capacidades para gestionarse a si mismos. Estas capacidades les permitirá adaptarse a los cambios en su entorno de ejecución. Mediante la autonomía, buscamos una reducción en el coste de operación y hacer más manejable la complejidad de los sistemas.

El sistema decide si es necesario ejecutar adaptaciones en base a directivas de alto nivel, los **objetivos**. Un operario humano define estas metas que el sistema debe alcanzar o mantener durante su ejecución. A partir de las políticas y la información del entorno, puede intuir que es necesario reconfigurarse para cumplirlas.

Para ello, cuenta con una serie de estrategias predefinidas que le permiten elegir su siguiente configuración. [12] Las adaptaciones pueden aplicarse de distintas formas: cambios en los parámetros de configuración, habilitar o deshabilitar funcionalidades, etc. Esto conlleva mover a tiempo de ejecución las decisiones de arquitectura y funcionalidad. Con ello, buscamos permitir un comportamiento dinámico del sistema. [3]

Siguiendo con el ejemplo de la tienda *on-line*, el operario podría definir un umbral máximo de carga por cada instancia. Cuando se supere, el sistema podría decidir que se requiere una acción correctiva. Por ejemplo, esta acción podría consistir en desplegar nuevas instancias del servicio. Cuando la carga de los servicios baje, podrá optar por eliminarlas.

Bucles de control

Para implementar estas capacidades de adaptación se recurre a la teoría de control y al **bucle de control** (o *feedback loop*). [3] Se trata de un proceso iterativo para la gestión de sistemas. A partir de información sobre el estado del sistema y su entorno, pauta acciones correctivas. Estas se basan en heurísticas definidas por los administradores del sistema. Puede dividirse en cuatro etapas (figura 2.3):

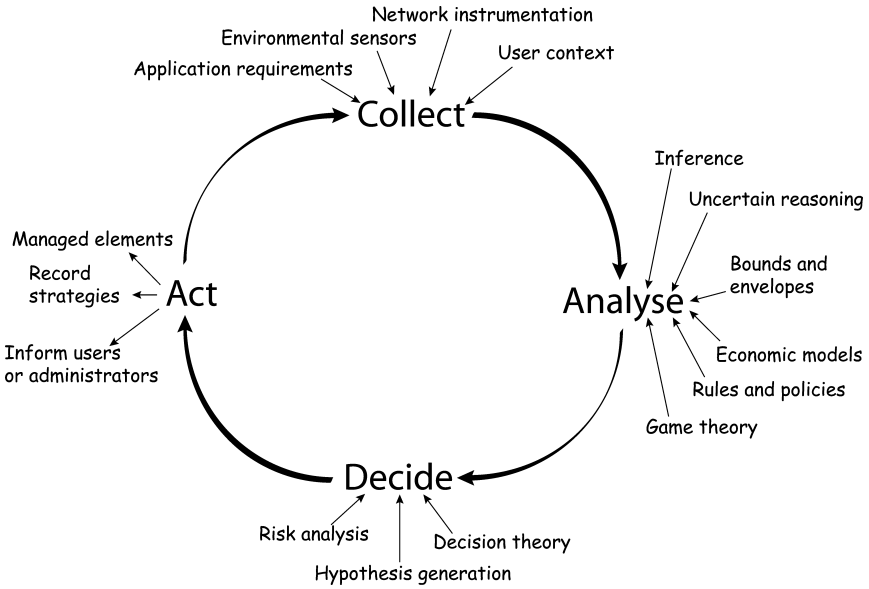


Figura 2.3: Un bucle de control genérico. Consta de cuatro actividades: Recopilar información, analizarla, decidir y actuar si procede. Obtenida de [13].

- **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 de rendimiento, estado de los componentes, cambios en en el entorno, etc.

  Estos datos en bruto deben ser limpiados, filtrados y agregados para sintetizarlos en propiedades de nuestro interés. Si se considera que son relevantes, se almacenan para informar las siguientes etapas del bucle.

- **Analizar**: Basándose en la información considerada de interés, la etapa de análisis debe identificar **síntomas**: indicadores de una situación que requiera de nuestra atención. Puede ser mediante heurísticas predefinidas, análisis estadístico u otros métodos. Un ejemplo de síntoma sería ”uso de CPU elevado”, ”número elevado de mensajes encolados en un sistema de mensajería”, etc.

- **Decidir**: A partir de los síntomas, el bucle debe determinar si es necesario tomar alguna acción correctiva. Podría detectarse que no estamos cumpliendo los objetivos, o que puede optimizarse la configuración actual. Para ello, se **planifica** qué acciones deben llevarse a cabo para que el sistema se adapte y alcance una configuración deseable. Por ejemplo, si hay muchos mensajes encolados, se solicitaría iniciar otra instancia del servicio que los consuma y procese en paralelo.

- **Actuar**: Si se ha planificado alguna acción se intentará **ejecutar** en esta etapa final. Mediante **efectores** en el sistema, el bucle es capaz de modificar su configuración. Dependiendo del éxito de ejecución, la adaptación se lleva a cabo o no. Finalizada esta etapa, se vuelve a recopilar información e inicia de nuevo el proceso.

En la ingenieria de *software*

En la ingeniería de *software*, los bucles de control suelen suelen implementarse de dos formas distintas: **implícitos** o **explícitos**. La más habitual es la primera: se encuentran implícitos en la implementación de los procesos del sistema. [3] No son componentes externos dedicados. Esto dificulta su implementación y mantenimiento ya que están entrelazados con la funcionalidad.

Por otro lado, aproximaciones como las de [2] o [12] optan por la segunda: bucles como componentes externos. Esto permite separar la funcionalidad de las capacidades de adaptación. Al dividirse estas responsabilidades, se puede reducir la complejidad de la implementación. En este trabajo nos centraremos en esta segunda variante.

En el caso de los bucles externos, pueden categorizarse además en base al **nivel** en el que operan. [14] Esto determinará el nivel de abstracción que tienen sobre el sistema que controlan, afectando a su reusabilidad en otras arquitecturas. De menor a mayor nivel de abstracción (y de reusabilidad) tenemos: nivel del sistema, mixto e infraestructura.

En el **nivel de sistema**, el bucle de control es un componente que se despliega al mismo nivel que el sistema manejado. Así, tendrá mucho más conocimiento de la solución y podrá ofrecer adaptaciones específicas para ella. Esto implica que acaba acoplado a ella y es menos reusable.

Por otro lado, en el **nivel de infraestructura**, el bucle se encuentra en un nivel de abstracción superior al sistema manejado. No tiene conocimiento sobre su implementación específica. Solo expone una serie de adaptaciones genéricas aplicables según la infraestructura en la que corre. Finalmente, el **nivel mixto** es una mezcla de ambas aproximaciones. El bucle tendrá componentes en ambas capas, capaces de comunicarse entre ellas para ofrecer una mejor capacidad de adaptación.

En cuanto a aplicaciones prácticas, podemos encontrarlos en gran variedad de contextos: balanceadores de carga [15], operación de plantas industriales [16], etc. Uno de los campos en lo que está teniendo más impacto es en el Internet de las Cosas (IoT). [17] En él, cada uno de los elementos debe operar de forma autónoma y ser capaz de colaborar con el resto de elementos de la red para cumplir con un objetivo común.

2.3 Arquitecturas para sistemas autónomos: Bucles MAPE-K

Un estilo arquitectónico muy representativo es el basado en bucles MAPE-K [2, 4] propuesto por IBM. Se trata de una referencia arquitectónica para desarrollar sistemas distribuidos autónomos. Nace con el objetivo hacer más manejable la complejidad de

estos sistemas; y reducir sus costes de operación, requiriendo de la minima intervención humana.

Sus componentes principales son los **elementos autónomos**. Cada uno de ellos es capaz de autogestionarse y colaborar con el resto de elementos del sistema para alcanzar los objetivos. Podría considerarse como una arquitectura basada en agentes. [17] A su vez, los elementos autónomos pueden dividirse en dos partes: un recurso manejado y un manejador autónomo (el bucle de control).

Los **recursos manejados** son las unidades de funcionalidad. Puede ser cualquier tipo de recurso, *hardware* o *software*. Para dotarlos de capacidad de autoadaptación, los emparejamos con un **manejador autónomo**: el bucle de control. Gestiona al recurso en base a la información que recoge del entorno de ejecución y las políticas que guían su adaptación.

El bucle es de tipo externo, ya que es un componente distinto al que implementa la funcionalidad. Por tanto, el recurso debe implementar puntos de contacto (***touchpoints***): interfaces que permitan obtener información de su estado (sondas) y cambiar su configuración (efectores).

Estos elementos autónomos se auto-gestionan en base a **políticas**: un conjunto de objetivos de alto nivel definidos por sus administradores. El sistema tratará de mantener su cumplimiento durante su ejecución. Para alcanzarlos, el manejador autónomo planifica cambios en la configuración del recurso manejado.

2.3.1. Estructura del bucle MAPE-K

En la figura 2.4 mostramos una representación de un elemento autónomo. Distinguimos las dos partes principales: el manejador y el recurso. El manejador contacta con el recurso a través de sus sensores y efectores. Podemos apreciar los componentes que conforman el bucle, y que describimos a continuación: [2]

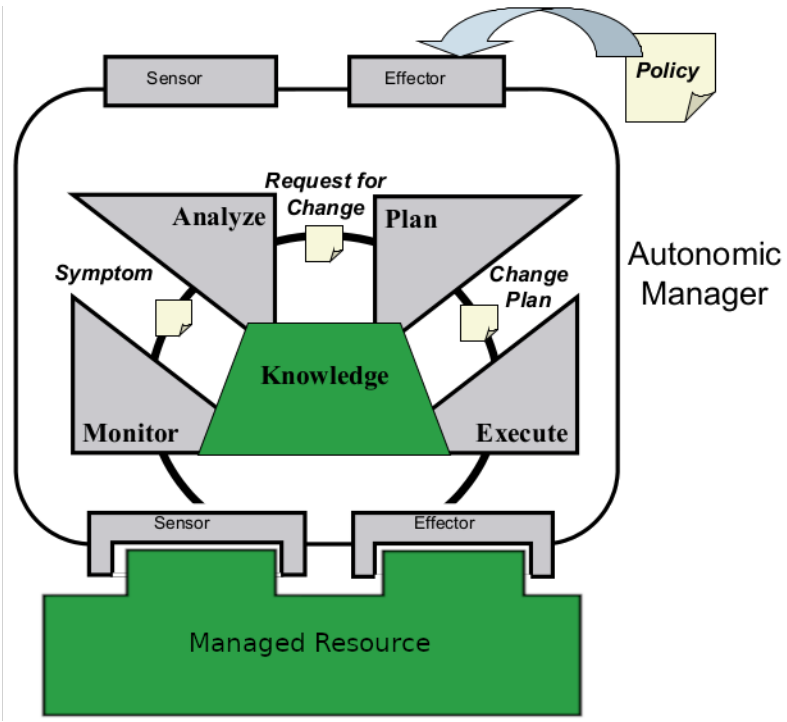


Figura 2.4: Representación de un elemento autónomo. Distinguimos el recurso manejado y el manejador autónomo. El manejador es un bucle MAPE-K. Basada en imagen de [2].

Para presentar estos componentes, describiremos un ejemplo de cómo se manejaría un servicio web. Nos centraremos en escalar este servicio en base a la carga del sistema. Deseamos que, en caso de carga elevada, se desplieguen nuevas instancias. Si la carga bajara, el sistema debería eliminar las instancias redundantes.

Sondas

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

Para nuestro servicio web, una métrica relevante sería el número de peticiones por segundo que está atendiendo. La sonda reportaría el número de peticiones que se han atendido hasta un determinado momento.

Monitor

El monitor recibe las mediciones de las sondas. Se encarga de recogerlas, agregarlas y filtrarlas para extraer información relevante. La información se almacenará como propiedades de adaptación en la base de conocimiento.[18] El monitor y las sondas componen la etapa de recopilar información de los bucles de control.

Siguiendo con nuestro ejemplo, el monitor recibiría el número de peticiones atendidas, y las agregaría en una métrica de serie temporal de peticiones por segundo. Esta sería una de nuestras propiedades de adaptación. En base a ella, las siguientes etapas tomarán las decisiones convenientes para escalar nuestro servicio.

Base de conocimiento

La base de conocimiento (*knowledge base*) es el componente base de toda la arquitectura. Informa a todas las etapas del bucle de control. Por lo que se trata de un componente transversal.

Está compuesta por una o más fuentes de información que el bucle tiene a su disposición. A partir de ellas, se almacenan las **propiedades de adaptación**. Estas describen el estado pasado y presente del sistema y su entorno: métricas, componentes, conexiones entre ellos, parámetros de configuración...

En conjunto, estas propiedades conforman un modelo abstracto del estado del recurso manejado que se mantiene en tiempo de ejecución. [12]. Las demás etapas del bucle operan en base a él. Como veremos más adelante, los efectores se encargan de traducir las acciones correctivas del modelo de alto nivel a términos del recurso manejado.

Analizador

En base al modelo abstracto del sistema, podemos razonar sobre el estado actual sin acoplarnos al recurso manejado. Podemos definir heurísticas que nos permitan detectar situaciones que requieran de una acción correctiva. Esta es la función del analizador.

Para implementarlo, una posible aproximación es mediante **reglas de adaptación**. Estas pueden dividirse en dos partes: la condición y la acción. La condición se define a partir de las propiedades de adaptación y evalúa si es necesario ejecutar la acción correctiva.

La acción de la regla describe una **propuesta de cambio** en la configuración del sistema. Estas se formulan en base a **operadores arquitectónicos**. [12] 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]

Las reglas se suscriben a cambios de las propiedades de las que dependen. Cuando ocurra alguno, se evalúa su condición. Si esta se cumple, se ejecuta la acción asociada. En caso contrario, no hará nada.

Respecto al servicio web, definiremos reglas tomando el valor del número de peticiones por segundo. Podemos definirlas con umbrales para este valor: si es muy alto, la regla solicita el despliegue de una nueva instancia. Cuando la carga baje, y si el servicio está replicado, podremos eliminarlas.

Planificador

Si alguna regla se dispara, el planificador recibe su propuesta de cambio. Este módulo se encarga de validar las acciones propuestas y agruparlas en un **plan de adaptación**. Para ello, recurre al conocimiento y compara el estado actual del sistema con las acciones solicitadas.

Deberá verificar si estas acciones siguen siendo necesarias. Podría ocurrir que desde que se solicitaron hasta que se genera el plan de adaptación, haya cambiado el estado del sistema. También comprobará si es seguro aplicarlas, ya que no deben dejar el sistema en un estado inconsistente.

Ejecutor

En la etapa final del bucle tenemos al ejecutor. Recibe el plan de adaptación del planificador y, como su nombre indica, es el encargado de ejecutarlo. Para ello, manipula los efectores del recurso manejado. Deberá identificar a cuáles debe transmitir el comando para realizar la adaptación.

Si una adaptación se lleva a cabo correctamente, deberá reflejarse en el conocimiento el nuevo estado, una vez se confirme. En caso de error, deberemos tener mecanismos de compensación que reviertan las acciones ejecutadas. Así, evitamos que el sistema quede en un estado inconsistente.

Efectores

Los **efectores** son el segundo tipo de *touchpoint* que debe ofrecer el recurso manejado. Ofrecen una interfaz común que permite al bucle modificar la configuración o estado del sistema. Deberán interpretar estas acciones, descritas en conceptos de alto nivel (nivel de arquitectura) y traducirlas a acciones de más bajo nivel (en términos del propio sistema). [12] Es decir, deberán determinar cómo ejecutarlas en el recurso manejado.

La comunicación entre este servicio y el sistema es un tanto especial: dependerá del sistema manejado; de si tenemos control sobre su implementación. Si no es así, tendremos que adaptarnos a la implementación que ofrezca este (HTTP, mensajería...).

En el caso del servicio web, la acción correspondiente sería desplegar o eliminar instancias. El efector conocerá el sistema de despliegue (p.e. Docker) y cómo solicitar la activación o desactivación de un servicio.

2.3.2. Sistemas distribuidos basados en elementos autónomos

Si nos fijamos en la figura 2.4, veremos que en la parte superior del elemento autónomo figuran sondas y efectores. Esto nos indica que pueden actuar también como recursos manejados, reportando mediciones y ofreciendo efectores para manipularlo. Nos permite colocar un manejador autónomo que actúe como **orquestador**. [2]

Los orquestadores gestionan uno o más elementos autónomos, responsabilizándose de tareas de más alto nivel. Facilitan también la cooperación entre sus elementos manejados. Por ejemplo, si nuestro elemento autonómico fuera un servidor web, el orquestador podría encargarse de gestionar varios servidores web distintos. Podría actuar como balanceador de carga u otros aspectos.

Por encima de los orquestadores tendríamos al administrador u **operario humano**. Como ya comentamos, este monitoriza el funcionamiento del sistema autónomo y lo gestionará mediante las políticas. Incluso puede participar en el proceso de toma de decisiones del bucle cuando este no cuenta con suficiente información para tomar una acción correctiva. Esto se conoce como ***human in the loop*** (humano en el bucle). [19].

La arquitectura final tendría el siguiente aspecto, mostrado en la figura 2.5.

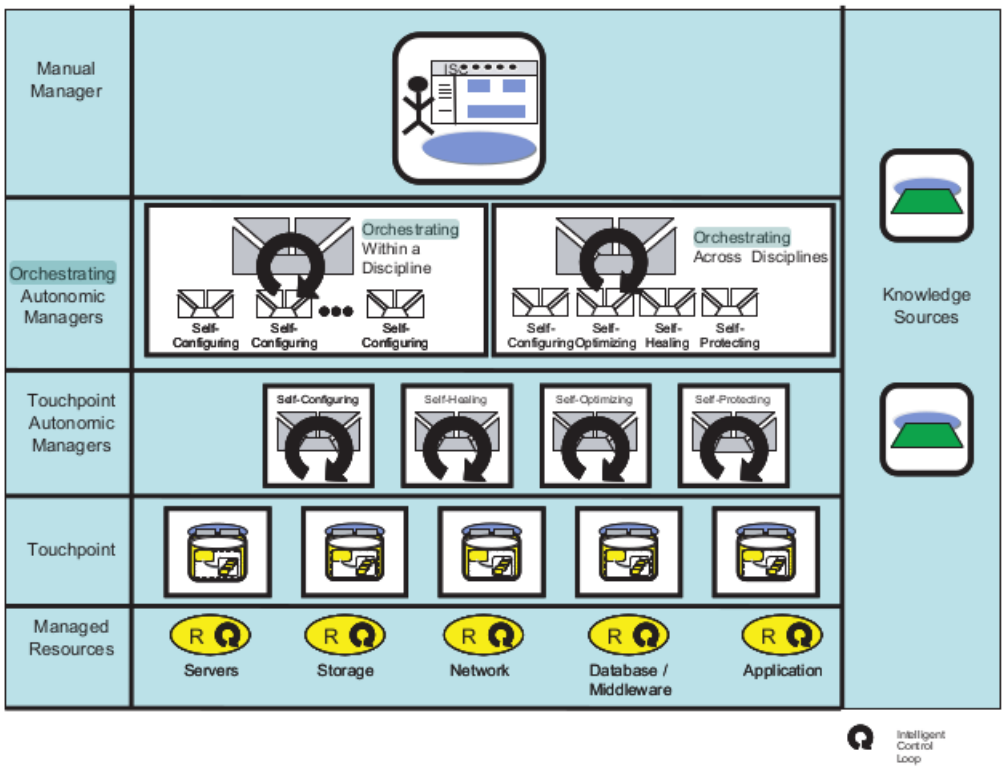


Figura 2.5: Arquitectura de un sistema autoadaptativo basado en MAPE-K. Imagen obtenida de [2].

CAPÍTULO 3
Sistema original

En este capitulo describiremos el sistema actual. Aquel que queremos dividir en microservicios. Exploraremos sus componentes y describiremos nuestros objetivos para dividirlo.

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 a cabo la migración; teniendo en cuenta las particularidades del sistema.

El servicio en cuestión implementa un **bucle de control MAPE-K**[2, 4], que ya describimos en la sección 2.3. Por suerte, partimos de un sistema cuyos componentes presentan una división funcional clara (cada etapa del bucle). Nos facilitará definir las fronteras de nuestros servicios.

Debido a esto, el foco de este capítulo pasará a los **conectores de *software***. Necesitamos establecer qué estrategias de comunicación utilizaremos para comunicar los servicios.

Buscar libros de descomposición de monolitos en microservicios.

CAPÍTULO 4
Diseño 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 a cabo la migración; teniendo en cuenta las particularidades del sistema.

El servicio en cuestión implementa un **bucle de control MAPE-K**[2, 4], que ya describimos en la sección 2.3. Por suerte, partimos de un sistema cuyos componentes presentan una división funcional clara (cada etapa del bucle). Nos facilitará definir las fronteras de nuestros servicios.

Debido a esto, el foco de este capítulo pasará a los **conectores de *software***. Necesitamos establecer qué estrategias de comunicación utilizaremos para comunicar los servicios.

Buscar libros de descomposición de monolitos en microservicios.

4.1 Distribución de los componentes

El primer paso fue identificar los componentes que compondrían nuestra arquitectura. Por suerte, partíamos de un sistema existente, con una arquitectura bien definida y documentada. Conocíamos el rol de cada uno de sus componentes 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? ¿Qué componentes debe abarcar cada microservicio?

En la figura 4.1 presentamos otra vista de la arquitectura actual del bucle. Una de las decisiones que tomamos muy temprano en el diseño fue separar cada etapa del bucle en su propio servicio. Esto nos aportaba varios beneficios:

El más evidente es que nos permitía independizar la implementación de cada etapa. Si fuera necesario, podríamos emplear distintas tecnologías para cada una. Incluso podríamos ofrecer implementaciones alternativas de algunos componentes. Estos podrían ofrecer distintas estrategias de la misma funcionalidad, como podrían ser distintos planificadores. TODO: cita sam newman

Por otro lado, también nos permitirá escalar cada etapa de forma independiente. Si por ejemplo el servicio de análisis estuviera recibiendo más peticiones que las demás, no sería necesario instanciar el bucle completo. En su lugar, podemos limitarnos a desplegar solo una nueva instancia del componente afectado.

Detectamos otra posible división de funcionalidad: actualmente, el bucle está muy acoplado al dominio de sus recursos manejados. Todo corre bajo el mismo proceso: el bu-

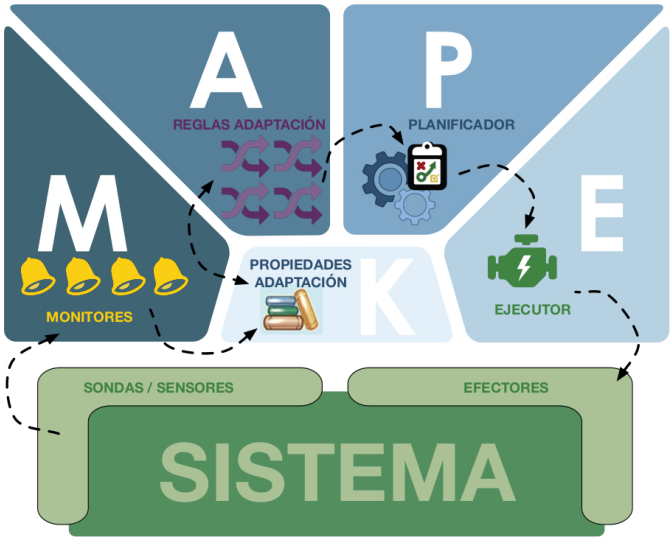


Figura 4.1: Arquitectura de un Bucle MAPE-K. El flujo de información y de control entre las etapas del bucle están representados con flechas. Obtenida de [18]

cle, los monitores, sus reglas de adaptación y demás elementos específicos de la solución. Ese proceso solo podrá manejar aquellos sistemas cuyos módulos tenga cargados.

Decidimos entonces desacoplarlos. Separar cada etapa del bucle MAPE-K de los elementos específicos de cada solución. Así, tendremos cada etapa como un microservicio agnóstico a una solución concreta; y por otro lado, estarán los servicios específicos para cada solución, con conocimiento del dominio del recurso manejado: monitores, reglas de adaptación, efectores...

Podremos entonces aprovechar la misma infraestructura para manejar varios sistemas simultáneamente (*multi-tennancy*). ampliar + cita

En cuanto al despliegue, mantendremos el bucle a nivel de sistema, [14] como funcionaba hasta ahora. Esto significa que se desplegará conjuntamente con los microservicios del recurso manejado. En la figura 4.2 mostramos los microservicios que componen nuestra arquitectura.

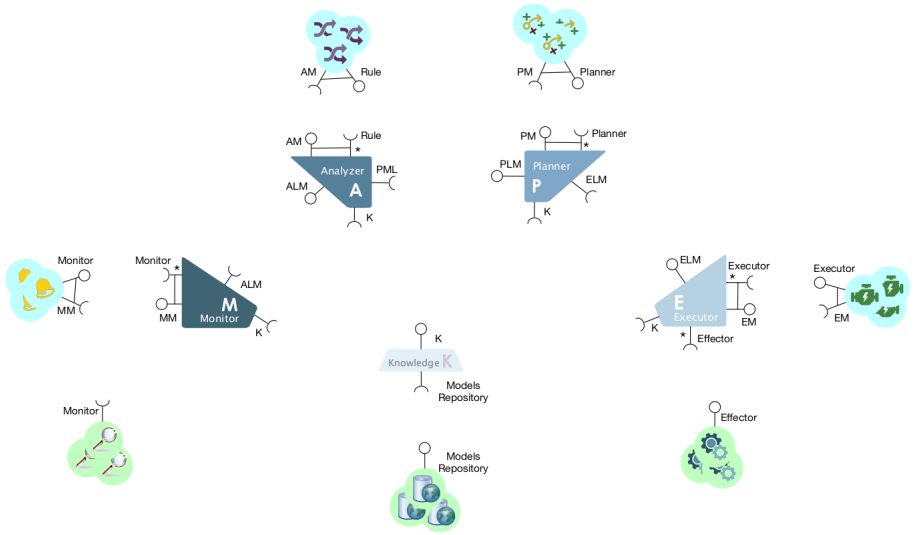


Figura 4.2: Diagrama con los componentes que forman nuestra arquitectura distribuida

Figura 4.2: Borrar los servicios específicos de planificador y ejecutor. Agrupar los servicios para poder aumentar zoom y hacerlo más legible. Añadir línea de divisón entre la capa del bucle y el dominio del recurso manejado.

4.2 Conectando los servicios

El siguiente problema al que nos enfrentamos está relacionado con la comunicación: si dividimos estos componentes en microservicios, ¿cómo deberían comunicarse? 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 arquitecturas existentes. 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*)[11, 20], en el que nos hemos inspirado.

4.2.1. 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 respecto al entorno de ejecución. En las capas inferiores, se encuentran los servicios más externos, más "acoplados". 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. Buscar ejemplo representativo

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 sobre el resto del sistema. Además, dentro de un mismo nivel no pueden contactar entre ellos. Según la dirección de la comunicación, se emplean mecanismos distintos (figura 4.3):

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

Basándonos en este estilo, definimos las capas de nuestro sistema. Esto nos permitió organizar los microservicios en jerarquías, lo que nos ayudaría a definir las reglas de nuestra arquitectura. Distinguimos cuatro niveles, de menor a mayor nivel de abstracción:

- **Nivel del recurso manejado**: En este nivel se encuentra el recurso manejado. Este implementa las sondas y efectores, los elementos que nos permiten interactuar con él. Hacen de intermediarios entre este y el resto del bucle, para reducir su acoplamiento.

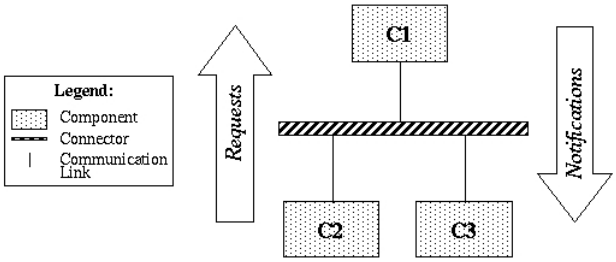


Figura 4.3: Ejemplo del estilo arquitectónico C2 (*Components and Connectors*). [20]

- **Nivel de solución**: En esta capa se encuentran componentes del bucle específicos para una solución concreta: monitores, reglas de adaptación, etc. No los incluimos en el mismo nivel que las sondas y efectores porque están a un nivel de abstracción distinto. Estos elementos guardan dependencia con el propio bucle de control.
- **Nivel del bucle**: Aquí se encuentran los servicios de las etapas del bucle: servicio de monitorización, análisis, planificación y ejecución. Esta capa es agnóstica al dominio de los recursos manejados. Además, actúa como intermediario entre los servicios de la solución y el conocimiento. Limitan cómo se accede 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 mayor nivel de abstracción. 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*. [21] 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, optamos por representarla como una arquitectura *knowledge driven* [5]. Nuestra capa central será la del conocimiento. A partir de ahí, cada nivel superior dependería de aquella a la que "envuelve": el bucle al conocimiento, la solución al bucle... En la figura 4.4 mostramos el resultado. Las flechas negras representan las peticiones, y las moradas, las notificaciones.

4.2.2. Definiendo los mecanismos de comunicación

Como comentamos antes, nos inspiramos en los mecanismos de comunicación descritos por C2: las peticiones y notificaciones. Pero, durante nuestra etapa de prototipado, nos dimos cuenta que 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 encuentran en la misma capa. Y, como dependen del conocimiento para funcionar, no podíamos moverlos a una capa superior para emplear las notificaciones.

Requeríamos por tanto de un tercer patrón de comunicación. Detectamos que este conector debería ser dirigido: los mensajes van dirigidos a un componente determinado.

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

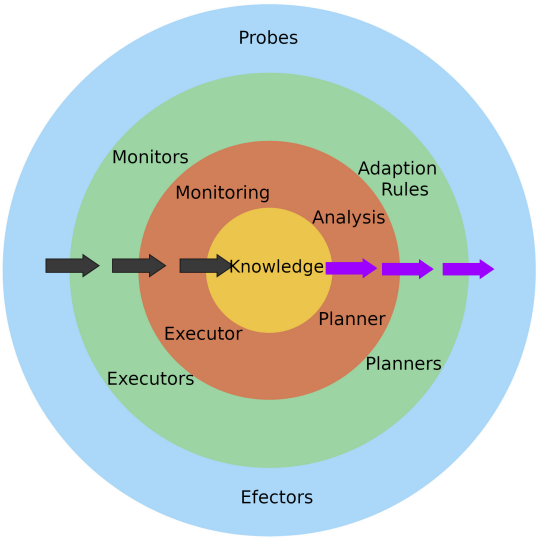


Figura 4.4: Representación de nuestra propuesta arquitectónica. Inspirado en Arquitectura Limpia (*Clean Architecture*). [1]

Además, como los elementos están en el mismo nivel, no queremos que se acoplen entre ellos. Surgió entonces la idea de utilizar las peticiones asíncronas, una combinación de los dos patrones existentes. A continuación lo presentamos junto al resto de patrones de comunicaciones que empleamos:

- **Peticiones síncronas**: Comunicaciones síncronas dirigidas a un servicio determinado. Un servicio contacta con otro con una petición o comando, y espera al resultado. Sólo están permitidas desde servicios de una capa más externa a un servicio en la capa interior adyacente.
- **Notificaciones**: Comunicaciones asíncronas no dirigidas. El servicio publica un evento que potencialmente recibirán todos los servicios en la capa externa adyacente. El cliente lo envía y continua su ejecución, sin esperar una respuesta.
- **Peticiones asíncronas**: Comunicaciones asíncronas dirigidas a un servicio determinado. Ideal para solicitar peticiones de trabajo asíncronas: se envían y el destinatario lo procesará cuando pueda. El cliente continuará su ejecución, sin esperar respuesta.

  Este mecanismo de comunicación solo está permitido entre elementos del mismo nivel. Para evitar el acoplamiento entre los componentes, deberemos buscar un conector que permita enviar el mensaje sin conocer específicamente al destinatario.

4.2.3. Conectores

Una vez determinadas las necesidades de comunicación de nuestro sistema, debemos buscar los conectores adecuados. Seguimos la estrategia descrita en [5] para elegir conectores; y nos basamos en los patrones de comunicación en sistemas distribuidos descritos en [22].

La estrategia consiste en centrarse en las comunicaciones entre cada par de componentes, y a partir de sus requisitos, determinar el tipo de conector adecuado. Sabiendo que hemos optado por una arquitectura distribuida, la elección de los **tipos de conecto-**

res se simplifica: los servicios pueden estar desplegados en máquinas distintas, por tanto el paso de mensajes será a través de la red.

Por ello, en lugar de recurrir a la taxonomía que lista [10], optamos por consultar las estrategias de comunicación habituales para sistemas distribuidos descritas en [22]. 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.

Hablar de smart endpoints, dumb pipes: https://simplicable.com/new/smart-endpoints-and-dumb-

Tipos de conectores

Invocación de métodos remotos o (***Remote Procedure Call***): Este 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 un resultado, si lo hubiera. Existen varios protocolos que implementan este mecanismo como gRPC o SOAP.

En la programación orientada a objetos suele emplearse una evolución de RPC: el paradigma de **objetos distribuidos**. [23] En este caso, el programa cliente puede interactuar con objetos que se encuentran 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. Por debajo, estos métodos realizan una llamada al servicio remoto donde se encuentra realmente. El servidor remoto procesa la petición y nos devolverá un resultado. En la figura 4.5 tenemos un esquema de este mecanismo.

Los *proxies* o (*stubs* en la terminología de RPC) suelen generarse a partir de un contrato que define qué operaciones ofrecen los objetos. Por ejemplo: SOAP con WDSL, gRPC; o en el caso de objetos distribuidos, Java RMI.

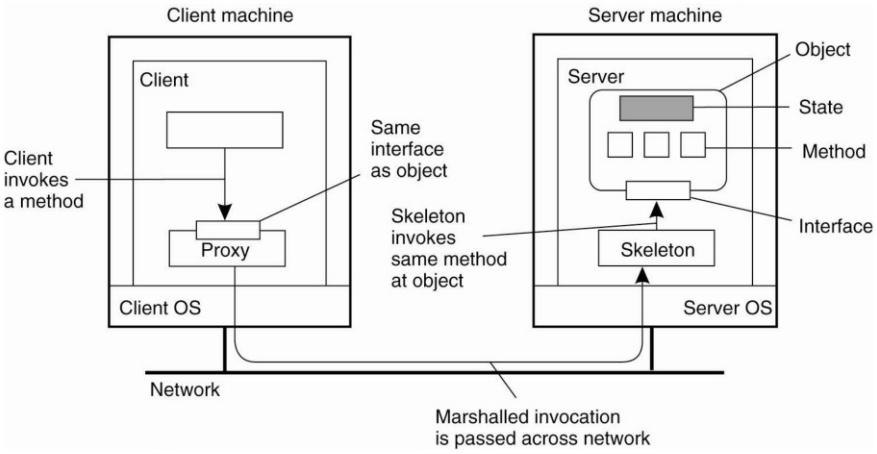


Figura 4.5: Funcionamiento del sistema de objetos distribuidos. Obtenido de [23]

- **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. Esto facilita la implementación.

- **Desventajas**:
  - **Dirigida**: Necesitamos conocer de antemano la ubicación del servidor al que queremos hacer una petición.
  - Dificulta la integración con otras aplicaciones. Cada servicio ofrece sus propias funciones. No están estandarizadas.
  - 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.
  - Respecto a los objetos distribuidos, 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. [24]
  - Si adoptamos sistemas como Java RMI, nuestro sistema se acopla a esa tecnología concreta. [22]. Nos quita flexibilidad en cuanto a qué otras tecnologías podemos emplear en nuestra arquitectura. ¿A Java o a Java RMI?

***Representational State Transfer* (REST)**: Se basa también en RPC, pero con ciertas restricciones adicionales. [5] Su concepto principal son los **recursos**: cualquier elemento sobre el que el servicio pueda ofrecernos información; y que pueda tener asociado un identificador único (una URI). [25] Por ejemplo, las entidades del dominio que gestiona nuestro servicio podrían ser recursos: 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 interfaz que pueden exponer los servicios REST es común. Solo cambia el "esquema de los datos", los tipos de recursos que sirven a los clientes. Esto facilita enormemente la integración con otros servicios. [26] La implementación más habitual es sobre el protocolo HTTP. Este 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**: Los métodos que exponen estos servicios están estandarizados y son sencillos. Un servidor solo debe implementar unos pocos métodos estándar que consumirán los clientes.
  - **Interoperabilidad**: Ampliamente utilizado en servicios de Internet. Es ideal para que clientes externos contacten con nuestro sistema mediante peticiones síncronas. [22]
  - **Comunicación síncrona**: Es el mecanismo ideal para comunicaciones síncronas. En ellas, el cliente requiere la respuesta del servicio para poder continuar con su procesamiento. Aunque también podemos dar soporte a comunicaciones *fire and forget*: el cliente envía un mensaje y no espera ninguna respuesta a su petición.

  - **Generación de clientes**: De forma similar a RPC, podemos generar clientes para facilitar la comunicación con APIs REST. Lo explicaremos con más detalle en la sección 4.2.4 cuando hablemos de OpenAPI.

- **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 binarios. 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 como métodos RPC (que no siguen REST).

GraphQL[2] AMPLIAR: Se trata de un protocolo para consultas de datos. Permite a los clientes ejecutar consultas personalizadas sobre los datos de un servidor. No se requiere de lógica específica para ejecutarla. De esta forma, el cliente puede obtener toda la información que necesita. Así se puede reducir 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 debamos obtener la misma información por composición, haciendo varias llamadas.

- **Desventajas**:

  - **Solo permite lecturas**: Es un lenguaje de consultas. No tiene comandos que permita escrituras.
  - **Exponemos datos a la red**: Se expone todos los datos a la red. Toda la información estará accesible para los clientes con permisos para acceder al EP.
  - **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, el *broker*, que actúa como intermediario. Gestiona la comunicación entre los servicios del sistema. [22] Existen varias estrategias de comunicación posibles: colas de trabajo, *publish-suscribe*, híbrida...

Tomemos por ejemplo las **colas de trabajo**. [27] 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

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

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 4.6 mostramos un ejemplo con dos consumidores (C1 y C2) a la escucha de la misma cola.

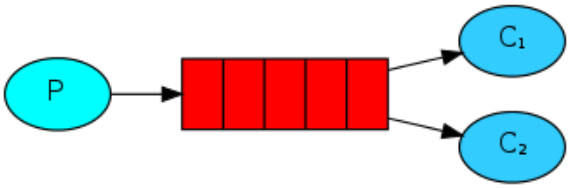


Figura 4.6: Representación de las colas de trabajo. Ejemplo de comunicación comunicación asíncrona dirigida. [3]

Otra estrategia posible es *publish-suscribe*: 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. [28] En la figura 4.7 tenemos un ejemplo. El mensaje "A" llega a todos los servicios suscritos a al tema "Topic".

Describir fanout Describir exchanges

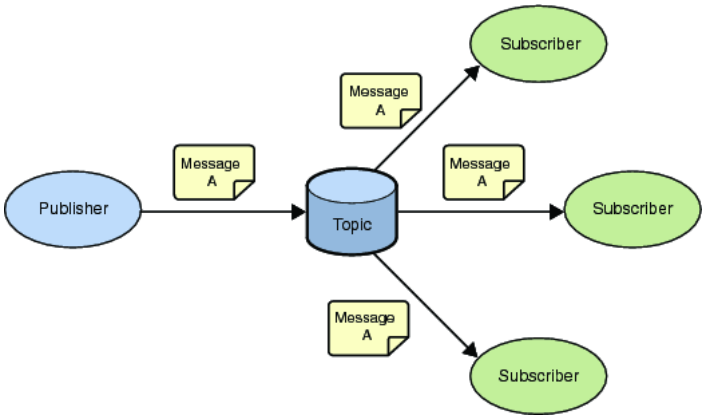


Figura 4.7: Estrategia *publish/suscribe*: el *broker* actúa como intermediario en la comunicación *multicast*. Imagen obtenida de [4].

La mayor ventaja de este estilo de comunicación es el **desacoplamiento** entre los servicios. [29] 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. Para enviar o recibir mensajes solo necesitan conocer su formato, las colas o temas y la dirección del *broker*.

- **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. Solo su formato, las colas o temas y la dirección del *broker*.

[3] Imagen obtenida de: https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html
[4] Java Messaging Service: https://docs.oracle.com/cd/E19509-01/820-5892/ref_jms/index.html

- **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. Este puede convertirse en un punto de fallo único. Para operar de forma fiable, estos sistemas requieren de replicación. [22]
  - **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 4.1 presentamos un resumen de esta comparativa:

	RPC	**REST**	**GraphQL**	**Broker mensajería**
Tipo de comunicación entre componentes	Dirigida	Dirigida	Dirigida	Dirigida y *Multicast*
Acoplamiento entre componentes	Alto	Medio	Alto	Bajo
Interoperabilidad	Baja	Alta	Alta	Alta[5]
Comandos de lectura	Sí	Sí	Sí	Sí[6]
Comandos de escritura	Si	Sí	No	Sí
Comunicación síncrona	Sí	Sí	Sí	No
Comunicación asíncrona	No	Sí	No	Si

Tabla 4.1: Comparativa de los mecanismos de comunicación.

Ahora describiremos qué protocolo elegimos para cada mecanismo de comunicación.

4.2.4. Peticiones síncronas

Comenzamos investigando las peticiones síncronas. Tomemos por ejemplo la comunicación entre el servicio de monitorización (*monitoring service*) y el servicio de conocimiento (*knowledge servic*). Recordemos que el servicio de conocimiento almacena todas las propiedades de adaptación. El resto de servicios del nivel del bucle necesitan consultarlas y actualizarlas durante su funcionamiento. En la figura 4.8 representamos inicialmente ambos componentes y un conector, sin especificar de qué tipo será.

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.

Entre las cuatro opciones de tipos de conector 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. Aunque podría ser interesante para consultas más avanzadas, utilizar dos protocolos de comunicación en paralelo aumentaría la complejidad de la arquitectura.

[6] Depende de si tenemos control sobre los componentes que queremos integrar.

[6] Aunque no es el mecanismo ideal para lecturas. Se recomienda que los mensajes sean ligeros. Los otros protocolos serían más adecuados.

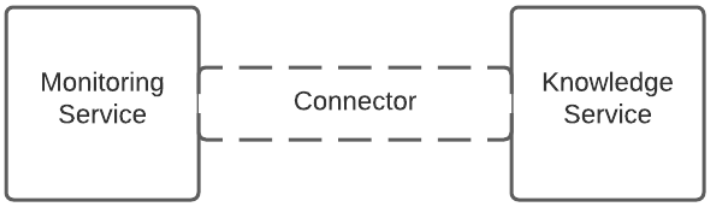


Figura 4.8: Representación inicial de la comunicación entre el servicio de monitorización y el de conocimiento. El conector no indica su tipo todavía.

También descartamos el *broker* de mensajería. Como requerimos de lecturas de datos, nos convenía más recurrir al resto de patrones. 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**: esta API estará expuesta "hacia fuera", a una capa más externa. Por tanto, tendremos clientes que nos contactarán. Potencialmente, de terceros. Prima por tanto la compatibilidad. Descartamos entonces RPC, dado que nos acoplaría a una tecnología concreta y a APIs no estándares.

Nos terminamos decantando por el conector REST sobre HTTP. Implementamos ambas funciones mediante *endpoints* HTTP. Su especificación se detalla a continuación en las tablas 4.2 y 4.3.

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: ▪ *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": "Celsius",`
	`  "ProbeId":"c02234d3-329c-4b4d-aee0-d220dc25276b",`
	`  "DateTime":"2022-01-15T18:19:38.5231231Z"`
	`  },`
	`  lastModification: "2022-01-15T18:19:39.123213Z"`
	`}`

Tabla 4.2: Especificación de la operación para obtener una propiedad del servicio de conocimiento.

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 *payload* 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 *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)`		

Tabla 4.3: 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 quedaba definir cómo se invocaría 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: generar clientes a partir del estándar OpenAPI.

Open API

OpenAPI[7] es un lenguaje estándar para describir APIs RESTful. Nos permite describir de forma estructurada las operaciones que ofrece un servicio 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 describe 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. [30] Más adelante mostraremos un ejemplo, con el fragmento 4.2.

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. Además, podemos aprovecharla en varios ámbitos del desarrollo gracias a la variedad de herramientas existentes: generación de documentación, generación de casos de prueba, identificar cambios incompatibles, etc. [31]

Uno de los casos de uso más interesantes es la generación de código a partir de la definición. Existen una serie de librerías[8] capaces de generar clientes o servidores conforme a la especificación. Ofrecen soporte a una gran variedad de lenguajes: Java, C#, JavaScript... En el caso del 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# usando 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, se podría obtener la especificación del prototipo y generar los clientes o servidores en cualquier lenguaje soportado. Java incluido. El bucle MAPE-K original después podría ser refactorizado usando este código autogenerado.

A continuación explicaremos cómo utilizamos OpenAPI. Para ello, continuaremos con el ejemplo del servicio de conocimiento que hemos descrito a lo largo de esta sección. Nos centraremos en la implementación de la operación para obtener una propiedad del conocimiento, descrita en la tabla 4.2.

En el fragmento 4.1 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. Si la encuentra, devuelve su valor con un código 200 OK. En ca-

[7] Página oficial: https://www.openapis.org/
[8] https://github.com/OpenAPITools/openapi-generator

so contrario, devuelve un código de error que describe el motivo (formato de la petición incorrecto o no se ha encontrado la propiedad).

Aparte de la implementación, podemos comprobar que el método cuenta 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 utilizarlos para generar una especificación más completa. Por tanto, resulta muy recomendable incluirlos.

```
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 4.1: 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 4.2, 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 4.2: 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*. Esta acepta determinados parámetros y describe unas posibles respuestas. También 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 4.1. Encontramos grandes similitudes con la especificación presentada en la tabla 4.2.

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 varias alternativas para solucionarlo: 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 (fragmento 4.3), optamos por implementar un *builder* de peticiones. Esto nos permitia configurar la petición de una forma más descriptiva para el usuario (fragmento 4.4):

```
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
```

```
var symptoms = new List<SymptomDTO> { new(SymptomName, "true") };

var systemConfigurationChangeRequest = new SystemConfigurationChangeRequestDTO
    ()
{
  ServiceConfiguration = changeRequests,
  Symptoms = symptoms,
  Timestamp = DateTime.UtcNow,
};

await _systemApi.SystemRequestChangePostAsync(
  systemConfigurationChangeRequest,
  CancellationToken.None);
```

Listing 4.3: Implementación de petición original. Trabajar con DTOs era muy verboso.

```
await _systemService.RequestChangeAsync(changeRequest =>
{
  changeRequest
    .ForSymptom(TemperatureGreaterThanHotThreshold)
    .WithService(ClimatisationAirConditionerConstants.AppName, service =>
    {
      service.MustBePresent()
        .WithParameter(
          ClimatisationAirConditionerConstants.Configuration.Mode,
          AirConditioningMode.Cooling.ToString());
    });
});
```

Listing 4.4: Implementación de la misma petición siguiendo el patrón *builder*.

Para terminar, mostramos la estructura del conector que emplearemos para implementar las peticiones aparece en la figura 4.9. 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*.

4.2.5. Notificaciones

El siguiente mecanismo de comunicación que tratamos fueron las notificaciones. Recordemos que esta comunicación consiste en transmitir mensajes desde un servicio a todos los que se pertenecen a la capa superior. Es una comunicación de tipo *multicast*. No va dirigida a ningún servicio concreto. Potencialmente, todos podrían recibir el mensaje y decidir si procesarlo o no.

Para estudiarla, tomamos como ejemplo la comunicación entre el servicio de conocimiento y los servicios en el nivel del bucle. Cada vez que se modifique una propiedad de adaptación o una clave de configuración, emitirá un evento a la capa superior. Así, por ejemplo, el servicio de análisis sabrá que debe reevaluar las reglas de adaptación.

Buscando el tipo de conector adecuado, empezamos descartando 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 cumplían nuestras necesidades. Tenemos el requisito de bajo acoplamiento contra los servicios de la capa superior. Estos protocolos requerirían de conocer la dirección de todos los servicios para poder contactarlos. O, en su defecto, requeriríamos de implementar un intermediario que los tenga registrados.

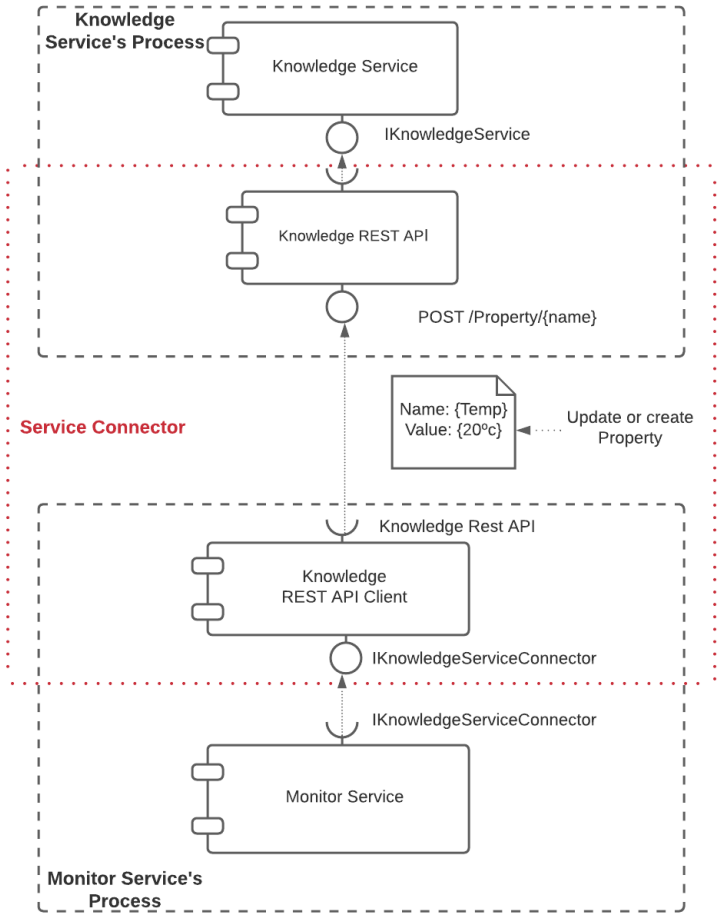


Figura 4.9: Diseño del conector usando implementación Cliente - Servidor

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* en un *exchange* determinado. Este evento tendrá como *topic* o tema asociado el nombre de la propiedad. Todos los servicios interesados deberán suscribirse a este. El *broker* añadirá el mensaje en la cola de cada uno de los suscriptores. Así podrán procesarlo cuando puedan, de forma asíncrona.

En la tabla 4.4 mostramos la especificación del evento. Podemos apreciar que el mensaje enviado incluye la información mínima indispensable: el nombre de la propiedad que ha cambiado. Tomamos esta decisión porque se trata de una comunicación asíncrona. Desde que se emite el evento hasta que se procesa su valor puede haber cambiado. De esta forma obligamos a las reglas a solicitar el valor de la propiedad en el momento en que se evalúen. Entonces siempre se ejecutaran con la información actualizada.

AsyncAPI

Para describir el evento, investigamos si existía algún estándar equivalente a OpenAPI. Y así es, se llama AsyncAPI[9]. Por desgracia, se encuentra en fases tempranas de su desarrollo. No ha alcanzado todavía el grado de implantación que tiene su homólo-

[9]Página oficial: https://www.asyncapi.com/

Evento	*PropertyChangedIntegrationEvent*	***Exchange***	*AdaptionLoop.Knowledge*
Descripción	Evento de integración que notifica sobre el cambio de una propiedad adaptación.		
Propiedades	*propertyName*	Nombre de la propiedad que ha cambiado.	

Ejemplo	Evento que notifica del cambio de la propiedad *Temperature*:
	`{`
	`  "PropertyName":"Temperature"`
	`}`		

Tabla 4.4: Especificación del evento que notifica sobre el cambio de una propiedad del conocimiento.

go. Por ejemplo, no cuenta un catálogo muy amplio de generadores de código. Tampoco podemos extraer la especificación a partir de una implementación existente.

Aun así, lo emplearemos para describir manualmente nuestros eventos en un formato estándar. En el fragmento 4.5 presentamos la especificación del mensaje de la tabla 4.4. Podemos apreciar similitudes con la especificación OpenAPI (por ejemplo, en el fragmento 4.2). Figura la estructura del mensaje y su documentación. La mayor diferencia es la mención del canal (el *exchange* en nuestro caso) y el método (*subscribe*). Esto indica que los consumidores podrán suscribirse a este evento a partir de este canal.

```
asyncapi: 2.4.0
info:
  title: Knowledge Service
  version: 1.0.0
  description: This service contains all the adaptation properties to inform
     the different stages of the loop.
channels:
  AdaptionLoop.Knowledge:
    subscribe:
      message:
        $ref: '#/components/messages/PropertyChangedIntegrationEvent'
components:
  messages:
    PropertyChangedIntegrationEvent:
      description: >-
        Integration event notifying about a change in an adaption property.
      payload:
        type: object
        properties:
          propertyName:
            type: string
            description: The name of the property that changed
```

Listing 4.5: Ejemplo del evento de integración *builder*.

Diseño del conector

Para implementar este patrón, nuestro conector estará compuesto por tres elementos: un publicador, el *broker* y un consumidor. Pongamos por ejemplo que el servicio de conocimiento recibe una petición para actualizar una propiedad. Si esta actualización se lleva a cabo, deberá propagar el evento a través del publicador. Este enviará el mensaje al *broker* a un *exchange* determinado.

El *broker*, que conoce todos los suscriptores, lo añadirá en la cola de mensajería de cada uno de ellos. Sus consumidores, desplegados en cada servicio suscriptor, serán no-

tificados del nuevo mensaje. Los procesarán en cuanto puedan. En la figura X mostramos la estructura de este nuevo conector.

TODO: Imagen del conector. Similar a 4.9

4.2.6. Peticiones asíncronas

El mecanismo de comunicación restante son las **peticiones asíncronas**. Se trata de peticiones de trabajo que un microservicio le envía a otro distinto. Ambos deberán encontrarse mismo nivel de la jerarquía. Como comentamos, tenemos dos casos en nuestra arquitectura que requieren de este patrón: la comunicación entre el módulo de análisis y el planificador; y aquella entre el planificador y el ejecutor. Nos centraremos en el primero.

Cuando se evalúan las reglas de adaptación, si alguna de ellas se ejecuta, propone un cambio en la configuración del sistema. Como estos servicios están en una capa superior a la del bucle, se lo transmiten al servicio de análisis mediante una petición síncrona. El módulo de análisis recibirá esta propuesta y la enviará al planificador mediante una petición asíncrona. Podríamos haberla enviado directamente directamente al servicio de planificación, pero preferimos que no se acoplen a un servicio adicional.

A la hora de escoger el mecanismo de comunicación, el razonamiento fue muy similar al empleado en las notificaciones. Optamos por implementarlas usando un *broker* de mensajería, pero siguiendo el patrón de colas de trabajo. Los *workers* cuentan con una cola de mensajería específica para las peticiones de trabajo. El publicador la conoce y, a través del *broker*, envía los mensajes allí. El consumidor los irá recuperando y procesando en cuando esté disponible.

En la tabla 4.5 presentamos la especificación de la petición asíncrona para solicitar un cambio de configuración de sistema. Vemos que es muy similar a 4.4. La principal diferencia es que esta incluye mucha más información que el evento. Esto es debido a que tiene más en común con una petición síncrona. El consumidor recibirá todos los parámetros que necesita para ejecutar la petición. Aun así, en el momento de ejecución, deberá verificar que algunos de estos parámetros no hayan cambiado.

Nombre	*SystemConfigurationChangeRequest*	**Cola**	*AdaptionLoop.Planification.Requests*
Descripción	Petición que representa una propuesta de cambio de la configuración del sistema.		
Propiedades	*Timestamp*	Fecha y hora de la petición de cambio.	
	Symptoms	Colección de síntomas que la han desencadenado.	

	Configuration Requests	Colección peticiones de configuración de la propuesta de cambio. Cada una de estas está compuesta por: ▪ ***ServiceName***: Identificador del servicio cuya configuración queremos cambiar. ▪ ***IsDeployed***: Indica si el servicio debe estar desplegado o no en la siguiente configuración. ▪ ***Bindings***: Colección de conexiones que indican a qué otros servicios debe estar conectado (o no) en la siguiente configuración. ▪ ***ConfigurationProperties***: Colección de pares clave-valor que representan valores de su configuración que queremos actualizar.

Ejemplo	Solicitud de cambio del modo de un aire acondicionado a modo calefacción (*heating*). Los síntomas indican que fue desencadenada porque la temperatura era menor que un umbral determinado:
	`{`
	`  "Timestamp": "2022-06-19T16:38:30.6092751Z",`
	`  "Symptoms":[`
	`  {`
	`  "Name": "temperature-lesser-than-cold-threshold",`
	`  "Value": "true"`
	`  }`
	`  ],`
	`  "ConfigurationRequests": [`
	`  {`
	`  "ServiceName": "Climatisation.AirConditioner.Service",`
	`  "IsDeployed": true,`
	`  "ConfigurationProperties": [`
	`  {`
	`  "Name": "Mode",`
	`  "Value": "Heating"`
	`  }`
	`  ],`
	`  "Bindings": []`
	`  }`
	`  ]`
	`}`	

Tabla 4.5: Especificación de las peticiones de cambio de configuración del sistema.

Respecto a la especificación con AsyncAPI, las peticiones asíncronas no están soportadas todavía. A fecha de la redacción, el grupo se encuentra estudiando cómo implementarlas. . Su inclusión está propuesta para la versión 3.0.0 de la especificación. Como mencionamos anteriormente, el estándar todavía se encuentra en fases iniciales de su desarrollo.

[9]Discusión disponible en: https://github.com/asyncapi/spec/pull/594

Diseño del conector

En cuanto a sus componentes, el conector seguiría la siguiente arquitectura: dibujo de arquitectura. La arquitectura del conector es muy similar a la de las notificaciones: un publicador, un broker y un consumidor.

4.3 Diseño final

Añadir diagrama con el diseño final, mostrando el diseño de los componentes con todos los conectores.

CAPÍTULO 5
Implementación

Una vez descrito el diseño del sistema, llegamos a la etapa de implementación. Uno de los objetivos del trabajo era verificar que la arquitectura elegida era viable. Para ello, optamos por implementar un sistema autoadaptativo muy básico. Tanto el diseño como la implementación se desarrollaron de forma incremental. Gracias a esto, el diseño fue evolucionando según detectábamos nuevas necesidades o problemas que no resolvía nuestra arquitectura.

En este capitulo describiremos la implementación de los microservicios del nivel de conocimiento de conocimiento y del bucle. Dejaremos para más adelante, en el capitulo 6, la descripción de la implementación del nivel de solución y sistema manejado. En este capitulo ofrecemos una vista más concreta de la implementación, y las tecnologías empleadas. En el otro describiremos cómo encaja todo a nivel general y veremos cómo opera.

La implementación se llevo a cabo en 4 hitos distintos, cada uno correspondiente a una etapa distinta del bucle:

- **Hito 1 - Servicio de monitorización y conocimiento**
- **Hito 2 - Servicio de análisis y reglas**
- **Hito 3 - Planificador**
- **Hito 4 - Ejecutor y efectores**

5.1 Servicio de monitorización y conocimiento

En esta primera etapa desarrollamos el proceso de monitorización. Este abarca desde que la sonda realiza sus mediciones hasta que se graban en el conocimiento. Esto implicó implementar varios componentes: las sondas y monitores del caso de estudio (capitulo 6), el componente de monitorización del bucle MAPE-K y la base de conocimiento.

Para su desarrollo se optó por el lenguaje C# y el *framework* ASP.NET[1]. Este *framework* es específico para implementar servidores web de la plataforma .NET de Microsoft. Lo elegimos porque ya contábamos con experiencia de desarrollo en esta plataforma. Además de que soporta los principales sistemas operativos (Windows, Linux y Mac).

[1]Página oficial: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core

5.1.1. Peticiones síncronas

En este hito también se prototiparon los conectores para peticiones síncronas. Aquellos servicios que las soporten expondrán *endpoints* HTTP. Por ejemplo, el servicio de conocimiento expone aquellos que permiten recuperar o modificar propiedades de adaptación. En el fragmento 4.1 ya mostramos un ejemplo de su implementación.

Gracias al uso de la librería *Swashbuckle.AspNetCore*[2], pudimos generar su especificación en el estándar OpenAPI. Esto nos aportó dos cosas: una interfaz de usuario para interactuar con la API y la posibilidad de generar el API client.

Hablemos primero sobre la interfaz de usuario. La librería añade a nuestro servicio el *endpoint* `/swagger`. Accediendo a esta ruta, se nos servirá una interfaz con un listado de todas las operaciones que ofrece la API (figura 5.1). De cada petición nos muestra su documentación (la que añadimos en el código) y sus parámetros. Incluso nos permite ejecutarlas. De esta forma, los usuarios pueden investigar qué ofrece la API y hacer pruebas.

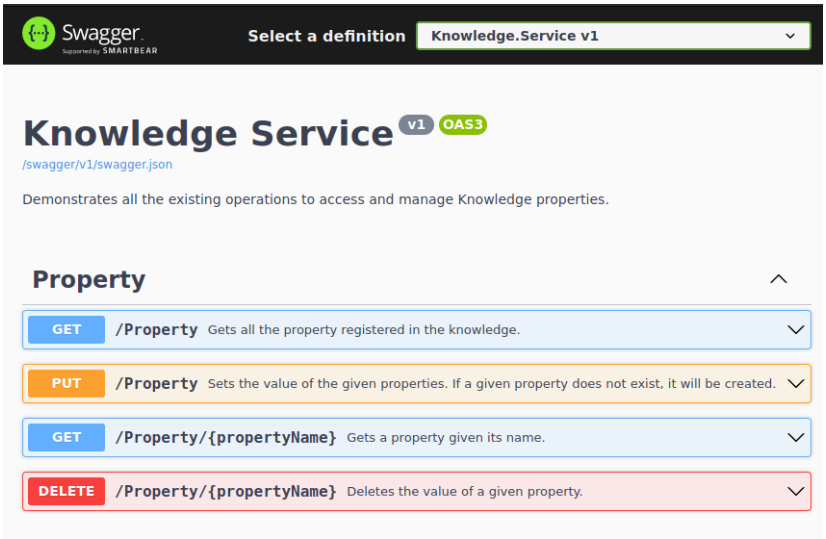


Figura 5.1: Interfaz de usuario ofrecida por Swagger para el servicio de conocimiento. Se genera a partir de las especificación OpenAPI.

Por otro lado, también nos permite generar el API Client. Como comentamos en la sección 4.2.4 de OpenAPI, tenemos gran variedad de generadores de código a nuestra disposición. Nosotros optamos por la librería `OpenAPI.Generator`[3]. En concreto, el generador de código de `C#`. Usándolo pudimos generar una librería que permite contactar con nuestro servicio, sin necesidad de implementar mucho código. Por ejemplo, el componente de monitorización del bucle contacta a través de un API Client generado.

5.1.2. Componentes: Módulos de monitorización y conocimiento

El módulo del conocimiento es un servicio muy sencillo. Ofrece operaciones de lectura y escritura sobre las propiedades de adaptación y las claves de configuración de servicios manejados. Para el prototipo estas se almacenan en diccionarios en memoria. Cuando el servicio se reinicie, los valores se perderán.

Por encima de este, tenemos el servicio de monitorización. En nuestra implementación, actúa como intermediario entre los monitores de la solución y el conocimiento.

[2]Página oficial: https://github.com/domaindrivendev/Swashbuckle.AspNetCore
[3]Página del proyecto: https://github.com/OpenAPITools/openapi-generator

También se trata de un servicio sencillo. Ofrece operaciones útiles para los monitores de la solución: lectura de propiedades del conocimiento y otras para que los monitores reporten sus mediciones. De esta forma, los monitores de solución podrán informarse a partir del conocimiento para determinar si una medición es válida o no.

¿Añadir ejemplos de código? ¿Describir los componentes implementados? ¿Debería unificarse con el capítulo de implementación del caso de estudio?

5.2 Servicio de análisis y reglas

En el segundo hito, acordamos implementar la evaluación de reglas de adaptación. Esto requería de implementar el servicio de análisis del bucle MAPE-K y los servicios de reglas de la solución. En este hito empezamos a plantearnos el diseño de las comunicaciones ascendentes: las notificaciones. Con ellas, evitaríamos que se los componentes se acoplaran a la capa superior.

5.2.1. Notificaciones

Comenzaremos describiendo el desarrollo de las notificaciones. Como ya se describió en la sección 4.2.5, este componente se implementó mediante un *broker* de mensajería. Elegimos `RabbitMQ`[4], uno "sencillo" y ampliamente utilizado. [22] Con él pudimos implementar los dos patrones de comunicación que necesitamos: las notificaciones y las peticiones asíncronas.

Para implementar nuestro conector, utilizamos una librería llamada `Rebus`[5]. Esta nos permite interactuar con un bus, abstrayéndonos de la tecnología concreta utilizada para la comunicación. Así, podríamos cambiar de tecnología de transporte en cualquier momento por otra que se ajuste más a nuestros requisitos.

Finalmente, para desacoplar la funcionalidad de de la publicación y consumición de mensajes del bus, empleamos `MediatR`[6]. Esta librería implementa el patrón mediador. Nos permite propagar mensajes dentro de un proceso. En nuestro caso, los eventos. Ni el emisor ni el receptor requieren tener referencias del otro. Es similar a un *broker* de mensajería, pero funcionando interproceso.

Ahora nos centraremos en la comunicación entre el módulo de conocimiento y el servicio de análisis. Una vez se confirma la escritura de una propiedad o configuración en el conocimiento, este debe notificar a los servicios en la capa superior. Para ello, comienza propagando internamente un **evento de integración** (línea 11 del fragmento 5.1) mediante el mediador.

```
1  private async Task SetProperty(SetPropertyDTO propertyDto)
2  {
3      var newValue = new()
4      {
5          Value = propertyDto.Value,
6          LastModification = DateTime.UtcNow,
7      };
8
9      properties.AddOrUpdate(propertyDto.Name, newValue, (_, _) => newValue);
10
11     await _mediator.Send(
12       new PropertyChangedIntegrationEvent(propertyDto.Name));
```

[4]Página oficial: https://www.rabbitmq.com/
[5]Página oficial: https://github.com/rebus-org/Rebus
[6]Página oficial: https://github.com/jbogard/MediatR

```
13 }
```

Listing 5.1: Implementación del método que asigna valor a una propiedad. Muestra un ejemplo de propagación interna de eventos de integración.

El mediador determinará que nuestro componente publicador es el destinatario de este evento. Lo hace en base a las interfaces que implementa (linea 2 del fragmento 5.2). Este componente publica el evento en el bus de mensajería (línea 15). Rebus, en base a la configuración del servicio, lo enviará a nuestra instancia de `RabbitMQ` y lo publicará en el *exchange* correspondiente. Todos los suscriptores ubicados en la capa superior lo recibirán en su cola.

```
1  public class PropertyChangedIntegrationEventPublisher
2    : IIntegrationEventPublisher<PropertyChangedIntegrationEvent>
3  {
4    private readonly IBus _bus;
5
6    public PropertyChangedIntegrationEventPublisher(IBus bus)
7    {
8        _bus = bus;
9    }
10
11   public async Task<Unit> Handle(
12       PropertyChangedIntegrationEvent notification,
13       CancellationToken cancellationToken)
14   {
15       await _bus.Publish(notification);
16
17       return Unit.Value;
18   }
19 }
```

Listing 5.2: El publicador de eventos captura el evento de integración y lo publica en el bus.

Finalmente, en el servicio de análisis, tenemos el componente consumidor (fragmento 5.3). `Rebus` obtendrá el evento de la cola de mensajería y se lo pasará a nuestro consumidor. Este lo recibirá y lo propagará internamente en el servicio (línea 20). Todos los manejadores (*handlers*) del evento lo recibirán y podrán tratarlo.

```
1  public class PropertyChangedIntegrationEventConsumer
2    : IIntegrationEventConsumer<PropertyChangedIntegrationEvent>
3  {
4    private readonly IMediator _mediator;
5
6    public PropertyChangedIntegrationEventConsumer(IMediator mediator)
7    {
8        _mediator = mediator;
9    }
10
11   public async Task Handle(PropertyChangedIntegrationEvent message)
12   {
13       await _mediator.Publish(message);
14   }
15 }
```

Listing 5.3: El consumidor recibe el evento de integración del bus y lo propaga internamente. Todos los manejadores de este evento lo recibirán.

5.2.2. Componentes: Servicio de análisis y módulos de reglas

En cuanto a la implementación del módulo de análisis, este realmente no cuenta tampoco con mucha lógica. Participa como intermediario entre el conocimiento y los ser-

vicios de reglas. Como hemos visto, recibe los eventos de cambios en las propiedades y los propaga a la capa superior. Ofrece operaciones de solo lecturas de propiedades y configuración del conocimiento. Así, limitamos las escrituras por parte de las reglas.

Se podría considerar que este servicio ha quedado anémico referencia. En trabajos posteriores, este servicio podría ampliarse añadiendo autenticación y autorización. Así, se podría evitar que servicios no autorizados accedan a sus operaciones o soliciten adaptaciones maliciosas.

Respecto a los módulos de reglas, ofrecemos una implementación de referencia. Aunque, como estos se encuentran en nivel de la solución, el desarrollador es libre de elegir si seguirla u optar por otra distinta.

En el fragmento 5.4 mostramos la clase base de las reglas de adaptación. Vemos que está suscrita a los eventos de integración de cambio de propiedad de adaptación y cambio en la configuración del sistema (lineas 2-3). Cuando el consumidor capture uno de ellos, lo propagará internamente. Todas las reglas afectadas lo capturarán.

Esta clase base se desarrolló siguiendo el patrón plantilla (o *template*)[7]. Define un método que evalúa la condición de la regla (`EvaluateCondition`) y, si esta se cumple, la ejecuta (`Execute`). Aquellas reglas que hereden de esta deberán de implementar ambos métodos.

```
1  public abstract class AdaptionRuleBase
2      : IIntegrationEventHandler<PropertyChangedIntegrationEvent>,
3        IIntegrationEventHandler<ConfigurationChangedIntegrationEvent>
4  {
5      // ..
6
7      private async Task Handle()
8      {
9          try
10         {
11             if (await EvaluateCondition())
12             {
13                 await Execute();
14             }
15         }
16         catch (Exception e)
17         {
18             _diagnostics.RuleEvaluationError(_ruleName, e);
19
20             throw;
21         }
22     }
23
24     protected abstract Task<bool> EvaluateCondition();
25
26     protected abstract Task Execute();
27
28     // ..
29 }
```

Listing 5.4: Clase base para implementar reglas de adaptación. Se evalúa la condición, y si esta se cumple, se ejecuta.

En cuanto a las suscripciones, las herederas deberán indicar de qué propiedades o claves de configuración dependen. En base a ellas, deberemos suscribirnos a los temas de las notificaciones que emite el módulo de análisis. Para ello, hemos implementado una

[7] https://refactoring.guru/design-patterns/template-method

serie de atributos que permiten declarar estas dependencias. Con ellos, decoraremos las clases

En el fragmento 5.5 mostramos un ejemplo. En la línea 1 tenemos el atributo que describe las dependencias con la propiedad de adaptación `Temperature`. Por otro lado, en las líneas 2-5 tenemos la declaración de dependencias con dos claves de configuración del servicio `Climatisation.AirConditioner`: `TargetTemperature` y `Mode`.

```
1  [RuleKnowledgePropertyDependency(ClimatisationConstants.Property.Temperature)]
2  [RuleServiceConfigurationDependency(
3      ClimatisationAirConditionerConstants.AppName,
4      ClimatisationAirConditionerConstants.Configuration.TargetTemperature,
5      ClimatisationAirConditionerConstants.Configuration.Mode)]
6  public class
       DisableAirConditionerWhenCoolingAndTargetTemperatureAchievedAdaptionRule
7    : AdaptionRuleBase
8  {
9      // ...
10 }
```

Listing 5.5: Las reglas declaran sus dependencias sobre propiedades de adaptación usando atributos. Estos se utilizarán para las suscripciones a los temas de los eventos.

Para que el servicio se suscriba a las notificaciones emplearemos la **reflexión**: analizaremos el ensamblado buscando todas las reglas y obtendremos los valores de sus atributos. En base a ellos, nos suscribiremos a los *topics* en el *broker* de mensajería.

```
1  public static IServiceCollection AddAdaptionLoopAnalysisServices(
2    this IServiceCollection services,
3    IConfiguration configuration,
4    Assembly rulesAssembly)
5  {
6      // ...
7
8      services.AddBus(
9          configuration,
10         rulesAssembly,
11         registerSubscriptions: async bus =>
12         {
13             var subscriptions = GetRulesBusTopicNames(rulesAssembly);
14
15             foreach (var subscription in subscriptions)
16             {
17                 await bus.Advanced.Topics.Subscribe(subscription);
18             }
19         });
20
21     return services;
22 }
```

Listing 5.6: Para suscribirnos a los *topics* de las notificaciones obtenemos las dependencias de las reglas mediante reflexión.

5.3 Planificador

En el tercer hito implementamos las peticiones de cambio de las reglas y el servicio de planificación. Aquí surge también la necesidad del tercer patrón de comunicación: las peticiones asíncronas.

5.3.1. Peticiones asíncronas

Continuando con las reglas de adaptación, llega el turno de describir su ejecución. Esta tiene lugar una vez se evalúa la condición y se estima que es necesaria una acción correctiva. En el cuerpo de la regla se describe la acción como una serie de cambios a la configuración del sistema. Para implementar el método, requerimos de un mecanismo para solicitar los cambios.

Las reglas comunicarán la solicitud al módulo de análisis mediante una petición síncrona. En el fragmento 5.7 mostramos un ejemplo. Usando su API Client, implementamos un *builder*[8] para simplificar la creación de solicitudes de cambio. En este ejemplo, indicamos cuál debería ser la siguiente configuración para el servicio `Climatisation AirConditioner Service` (líneas 7-14). Deberá estar activo (lineas 7-10) y su propiedad `Mode` con el valor `Cooling` (lineas 11-13). En esta petición se incluye también el síntoma que desencadena el cambio (línea 6).

```
1  protected override async Task Execute()
2  {
3    await _systemService.RequestConfigurationChange(changeRequest =>
4    {
5      changeRequest
6        .ForSymptom(TemperatureGreaterThanHotThreshold)
7        .WithService(ClimatisationAirConditionerConstants.AppName, service =>
8        {
9          service
10           .MustBePresent()
11           .WithParameter(
12             ClimatisationAirConditionerConstants.Configuration.Mode,
13             AirConditioningMode.Cooling.ToString());
14       });
15   });
16 }
```

Listing 5.7: Implementación de la misma petición siguiendo el patrón *builder*.

El servicio de análisis recibirá la petición y la redirigirá al planificador. Lo hará mediante una petición asíncrona. Su implementación es muy similar a la de las notificaciones, explicada en detalle en el apartado anterior. La mayor diferencia radica en la cardinalidad de la comunicación: el mensaje se publicará directamente en la cola de trabajo, en lugar de publicarlo en un exchange. Es decir, como mucho lo procesará un solo servicio.

Esto implica que sólo cambiará la implementación del publicador. En la línea 14 del fragmento 5.8 vemos que el mensaje se enruta directamente a la cola `PlanningServiceQueue`. El resto de la implementación será muy similar. Solo cambiará las interfaces que debamos implementar para cada tipo de componente (`IRequestConsumer` en lugar de `IIntegrationEventConsumer`, etc.).

```
1  public class SystemConfigurationChangeRequestPublisher
2    : IRequestPublisher<SystemConfigurationChangeRequest>
3    where TRequest : Request
4  {
5    public SystemConfigurationChangeRequestPublisher(IBus bus)
6    {
7        _bus = bus;
8    }
9
10   public async Task<Unit> Handle(
11       SystemConfigurationChangeRequest request,
12       CancellationToken cancellationToken)
```

[8] Patrón *builder*: https://refactoring.guru/design-patterns/builder

```
13    {
14        await _bus.Advanced.Routing.Send(
15            AdaptionLoopPlanningConstants.Queues.PlanningServiceQueue,
16            request);
17
18        return Unit.Value;
19    }
20 }
```

Listing 5.8: Las peticiones asíncronas se publican a una cola determinada.

5.3.2. Componentes: Servicio de planificación

El planificador recibirá esta petición de cambio y deberá elaborar un **plan de adaptación**. Para ello, verificará que la solicitud es viable. En nuestro prototipo, para reducir el alcance del proyecto, nos limitamos a comprobar que la configuración sea distinta a la actual. Añadirá al plan las **acciones de adaptación** requeridas para alcanzar el estado deseado. Si el sistema ya estuviera en ese estado, el plan de cambio se queda vacío y no se propaga.

Ya comentamos que los operadores arquitectónicos con los que operaremos serán del ámbito de los microservicios: activar servicio, eliminar servicio, vincular servicios, desvincularlos y cambiar su configuración.

Por ejemplo, en el fragmento 5.9 encontramos un plan de adaptación para la regla descrita en la sección anterior. Solo contiene una acción de adaptación: cambiar el valor de la propiedad `Mode` a `Cooling`. Como el servicio de aire acondicionado ya estaba en funcionamiento, no se ha incluido una acción para desplegarlo.

```
{
  "ChangePlan": {
    "Timestamp": "2022-07-09T09:53:01.1868834Z",
    "Actions":
    [
      {
        "Type": "SetParameter",
        "ServiceName": "Climatisation.AirConditioner.Service",
        "PropertyName": "Mode",
        "PropertyValue": "Cooling"
      }
    ]
  },
  "Symptoms":
  [
    {
      "Name": "temperature-lesser-than-cold-threshold",
      "Value": "true"
    }
  ]
}
```

Listing 5.9: Plan de adaptación generado para la regla anterior. Solo contiene una acción de adaptación: cambiar la configuración `Mode` del servicio `AirConditioner`.

5.4 Ejecutor y efectores

En el hito final implementamos el módulo ejecutor y los efectores. Cerramos así el ciclo del bucle de adaptación. El ejecutor recibe el plan de adaptación del planificador. A

partir de este, deberá distribuir las acciones de adaptación que contiene entre los servicios de efectores.

Para transmitirle el plan de adaptación del planificador al ejecutor, volvemos a recurrir a las peticiones asíncronas. El plan de adaptación que se le pasa como evento de integración es el que ya mostramos en el fragmento 5.9.

Una vez captura la petición, el ejecutor deberá determinar qué servicios debe cambiar y manipular sus efectores. En nuestra implementación de referencia, simplemente agrupamos las acciones por el nombre del servicio afectado. Publicaremos cada grupo como notificaciones individuales, usando el nombre del servicio como tema.

Los servicios de efectores del recurso manejado las capturarán. Estos servicios procesan las acciones asignadas. Si contienen el efector correspondiente, la ejecutarán. Dependiendo del tipo de acción, el efector hará una acción u otra: desplegar un servicio, o eliminarlo, cambiar la configuración, etc.

En cuanto a la comunicación con el sistema, este caso es un tanto especial. El mecanismo dependerá del sistema manejado; de si tenemos control sobre su implementación. Si no es así, tendremos que adaptarnos a aquellos que ofrezca el recurso (HTTP, mensajería...).

Continuando con nuestro ejemplo del modo del aire acondicionado, el sistema modificaría la configuración del mismo. Una vez se confime el cambio, si se ha llevado a cabo, se ejecutará actualizará su valor en la base de conocimiento. Dependerá de si el sistema es capaz de hacerlo o debe recaer la responsabilidad en el efector.

CAPÍTULO 6
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.

6.1 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.

6.2 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 [18].

6.2.1. Sondas:

Para implementar el sistema, requerimos de las siguientes sondas:

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

Tabla 6.1: Sondas del sistema de climatización.

6.2.2. Propiedades de adaptación:

También podemos deducir cuáles son nuestras propiedades de adaptación:

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

Tabla 6.2: Propiedades de adaptación del sistema de climatización.

6.2.3. 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 5ºc 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:	*climatisation.monitor.temperature*
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:	*temperature*
Acciones:	**SI** | *new-temperature* - *temperature* | <= 5.0 **O** request.DateTime - previousMeasurement.DateTime >60s **ACTUALIZA-KNOWLEDGE** *temperature* = *new-temperature*
Monitor:	*climatisation.monitor.configuration*
Descripción:	Recibe la configuración del aire acondicionado y la registra en el *knowledge*.
Afecta a propiedades de adaptación:	*airconditioner.is-deployed*, *airconditioner-mode*, *target-temperature*, *cold-temperature-threshold*, *hot-temperature-threshold*
Acciones:	**SI** *property* != *new-value* **ACTUALIZA-KNOWLEDGE** *property* = *new-value*

Tabla 6.3: Monitores del bucle MAPE-K del sistema de climatización.

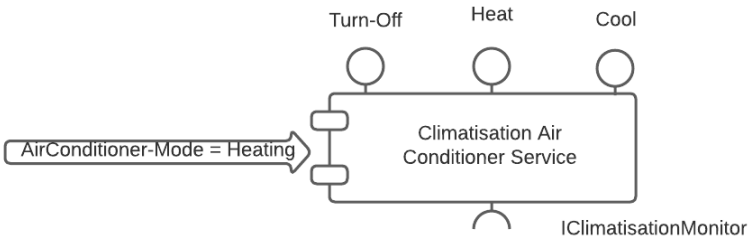
6.2.4. Reglas de adaptación

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.

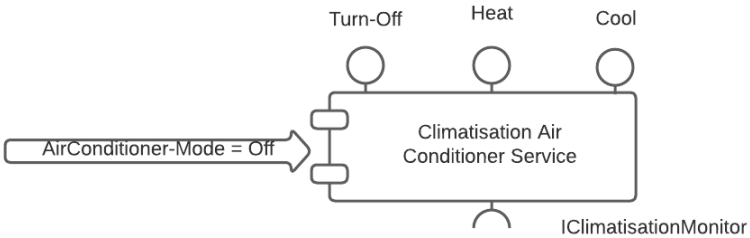
Como comentamos en el capitulo anterior, en nuestro ejemplo de bucle MAPE-K, nos limitamos a implementar las adaptaciones de tipo set-parameter. Por tanto, no tendremos reglas de despliegue o de binding.

En la tabla 6.4 definimos las cuatro reglas necesarias:

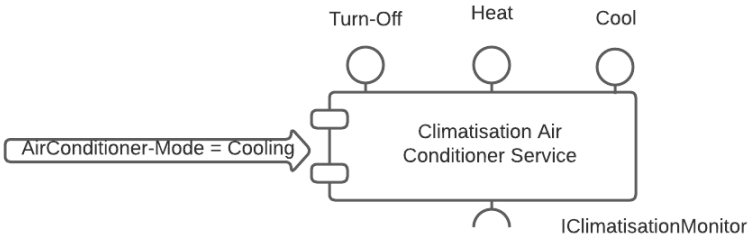
Regla:	*EnableAirConditionerHeatingModeWhenColdTemperatureThresholdExceeded*
Descripción:	Activa el aire acondicionado en modo calefacción cuando la temperatura sea inferior al umbral de frío.
Condición:	*airconditioner-mode* != *Heating* **AND** *temperature* <= *cold-temperature-threshold*
Cuerpo:	



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

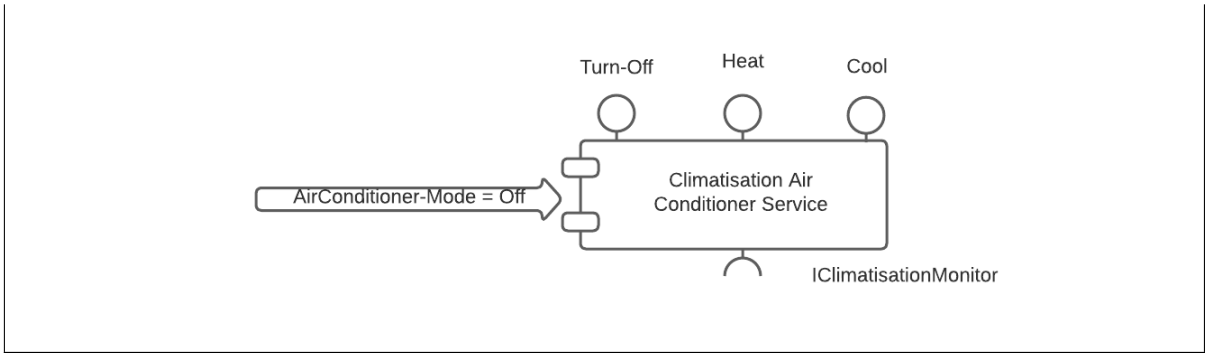


Tabla 6.4: Reglas de adaptación del sistema de climatización.

6.2.5. 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.

Efector:	*airconditioner.heat*
Descripción:	Activa el modo calentar del aire acondicionado.
Efector:	*airconditioner.cool*
Descripción:	Activa el modo enfriar del aire acondicionado.
Efector:	*airconditioner.turn-off*
Descripción:	Apaga el aire acondicionado.

Tabla 6.5: 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.

6.2.6. 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

6.3 Implementación

Para la implementación, hemos utilizado las mismas tecnologías descritas en el capítulo 5: microservicios `ASP.NET`, comunicación mediante APIs REST y *brokers* de mensajería `RabbitMQ`. Generamos los API Clients con OpenAPI y demás.

A continuación describiremos todos los servicios implementados:

6.3.1. Servicio de aire acondicionado

El servicio de aire acondicionado será nuestro sistema manejado. Como no disponemos de un aire acondicionado real, hemos optado por implementar uno ficticio. Este cuenta con tres estados de operación: apagado (`OFF`), enfriando (`COOLING`) o calentando (`HEATING`). Según el modo, aumentará o disminuirá la temperatura que reporta un termómetro interno. Además, cuando está apagado, la temperatura ficticia aumenta o disminuye gradualmente según una configuración. De esta forma, podemos simular los cambios de temperatura más rápido y ver si se aplican correctamente las adaptaciones correspondientes.

Además, este servicio expone tres endpoints HTTP, que nos permiten cambiar el modo del aire acondicionado. Serán estos los que invocará el servicio de efectores para manipular su estado.

Por otro lado, para simplificar la configuración del sistema, optamos por registrar las propiedades del aire acondicionado durante su arranque. Tenemos para ello un servicio en segundo plano (un *background service* de `ASP.NET`) que registra la configuración inicial durante el arranque. Así le daremos valor a las propiedades como los umbrales de temperatura o las variables como is-deployed. Es necesario un fragmento de código?

6.3.2. Monitor

El monitor de la solución expone endpoints para recabar las mediciones de las sondas. En este caso, las mediciones de temperatura del aire acondicionado: `POST Measurement/Temperature`. Este endpoint recibe la temperatura y ejecuta las validaciones descritas en el apartado anterior: descarta valores con diferencias de 5ºC tomados en menos de un minuto, entre otras. Si el valor es válido, lo manda al servicio de monitorización. A su vez este lo almacenará en el conocimiento como propiedad de adaptación.

El servicio también ofrece endpoints para actualizar la configuración del aire acondicionado en el conocimiento. Así podrá registrar su configuración inicial o los modos de operación cuando cambien. Son un subconjunto de los endpoints que ofrece el conocimiento, expuestos a través del servicio de monitorización del bucle.

6.3.3. Reglas

En cuanto al servicio de reglas, este contiene las cuatro reglas definidas en el apartado 6.2.4. Estas activan o desactivan el aire acondicionado en base a la temperatura de la estancia. Se han implementado siguiendo la estructura descrita en el apartado 5.2. Todas ellas heredan de las clase abstracta `AdaptionRuleBase`. Deben implementar los métodos para evaluar la condición y ejecutar su acción; y deben declarar las propiedades de adaptación de las que dependen.

Para describirlas, nos centraremos en la regla `Disable Air Conditioner When Cooling And Target Temperature Achieved Adaption Rule`. Esta desactiva el aire acondicionado cuando está en modo enfriamiento y se ha alcanzado la temperatura de enfriamiento de la estancia. A lo largo de la memoria ya habíamos mostrado algunos fragmentos de código de esta.

Las reglas describen las propiedades o configuraciones de las que dependen mediante atributos. Se trata de aquellas que requiere para evaluar su condición. Tomemos por ejemplo el fragmento 5.5. Observamos que declara de tres dependencias: `airconditioner-mode`, `temperature` y `hot-temperature-threshold`.

En cuanto a la implementación, ya habíamos mostrado su método `Execute` en el fragmento 5.7. Solo nos queda exponer la implementación de referencia del método `Evaluate Condition`. En el fragmento 6.1 ofrecemos la implementación de la condición *airconditioner-mode == Cooling* **AND** *temperature <= target-temperature*. Observamos que en las líneas 3-5, 12-15 y 17-20 se obtienen las propiedades de adaptación o claves de configuración desde el servicio de análisis. En base a ellas, en las líneas 22-23 se evalúa la condición descrita.

```
1  protected override async Task<bool> EvaluateCondition()
2  {
3    var currentTemperature =
4      await _propertyService.GetProperty<TemperatureMeasurementDTO>(
5        ClimatisationConstants.Property.Temperature);
6
7    if (currentTemperature is null)
8    {
9        return false;
10   }
11
12   var airConditionerMode = await _configurationService
13     .GetConfigurationKey<AirConditioningMode?>(
14         ClimatisationAirConditionerConstants.AppName,
15         ClimatisationAirConditionerConstants.Configuration.Mode);
16
17   var targetTemperature = await _configurationService
18       .GetConfigurationKey<float?>(
19         ClimatisationAirConditionerConstants.AppName,
20         ClimatisationConstants.Configuration.TargetTemperature);
21
22   return airConditionerMode == AirConditioningMode.Cooling
23       && currentTemperature.Value <= targetTemperature;
24 }
```

Listing 6.1: Implementación de referencia del método `EvaluateCondition`. La regla obtiene del conocimiento el estado actual del sistema y determina si debe ejecutarse.

6.3.4. Efectores

Finalmente, tenemos el servicio de efectores. En la última etapa del bucle de adaptación, el módulo de ejecución emite una notificación con las acciones de adaptación que debe ejecutarse para determinado componente del sistema manejado. El servicio de efectores se suscribirá a estas notificaciones de aquellos componentes que gestiona. En este caso, las referentes al aire acondicionado.

Una vez capturada, determinará el tipo de acción que debe ejecutar e invocará a su implementación. En nuestro caso, hacemos un dispatch interno del evento

Para el caso de estudio, por restricciones de tiempo, nos limitamos a implementar las adaptaciones que implicaban cambios en la configuración del sistema manejado (adaptaciones *set parameter*). La implementación del componente manejado determinará cómo se ejecutarán estas acciones. Dependemos de los efectores que expongan. Por ejemplo, para el servicio de aire acondicionado, cambiar el modo de operación supone invocar a unos endpoints que expone. Por ejemplo: (`POST /airconditioner/turn-off`).

```
1  public async Task<Unit> Handle(
2    SetAirConditionerModeRequest notification,
3    CancellationToken cancellationToken)
4  {
5      var succeeded = Enum.TryParse(
6        notification.Value,
7        out AirConditioningMode mode);
```

```
8
9       if (!succeeded)
10      {
11          return Unit.Value;
12      }
13
14      switch (mode)
15      {
16          case AirConditioningMode.Off:
17              await _airConditionerApi.AirConditionerTurnOffPostAsync(
                    cancellationToken);
18              break;
19
20          case AirConditioningMode.Cooling:
21              await _airConditionerApi.AirConditionerCoolPostAsync(
                    cancellationToken);
22              break;
23
24          case AirConditioningMode.Heating:
25              await _airConditionerApi.AirConditionerHeatPostAsync(
                    cancellationToken);
26              break;
27      }
28
29      return Unit.Value;
30  }
```

Listing 6.2: Implementación de los efectores del aire acondicionado. Invocan a los endpoints HTTP en base a las acciones de adaptación.

6.4 Despliegue y Pruebas

Debido a la gran cantidad de microservicios que componen la solución, optamos por empaquetarlos en contenedores `Docker`. Para ello, definimos un plan de despliegue con Docker Compose, que nos permitía definir el número de instancias y las dependencias entre ellas. Por ejemplo, todos los servicios que requieran de un bus de mensajería, dependen de la instancia de RabbitMQ desplegada. Por simplicidad, optamos por desplegar una única instancia para toda la solución. Lo ideal sería que cada que requiriera de emitir sus propias notificaciones dispongan de la suya propia, que se adapte más a sus necesidades.

6.4.1. 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 6.1. 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.

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 [32] y consta de tres partes distintas:

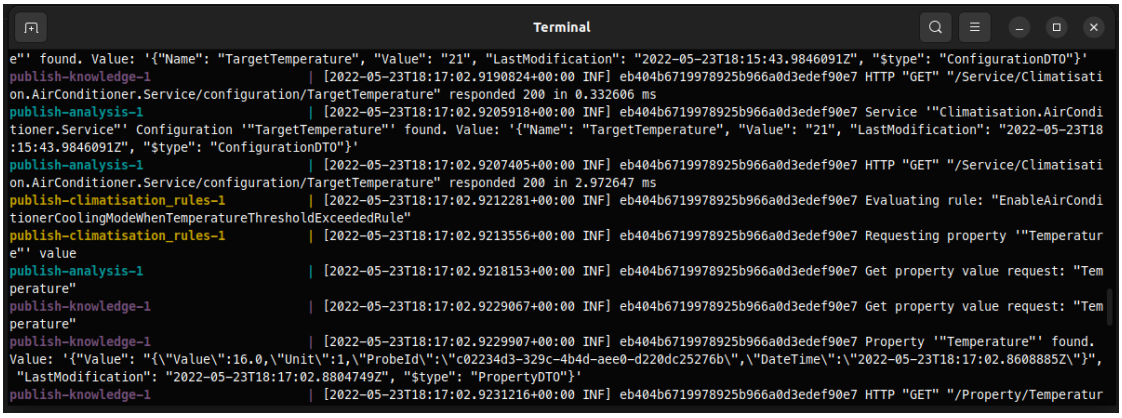


Figura 6.1: Extracto de *logs* de una ejecución habitual.

- **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. [33] 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. [32]
- **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. [33]
- **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 transverses process, network and security boundaries. [33] 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. [32]

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:

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.

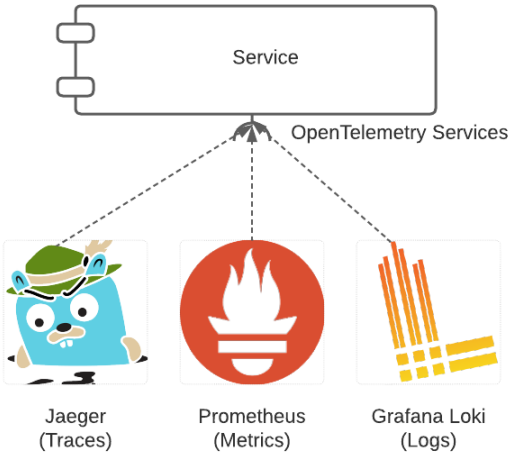


Figura 6.2: Extracto de *logs* de una ejecución habitual.

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 7
Conclusiones

7.1 Trabajos futuros

- Implementar mecanismos de autenticación y dar soporte a multitenancy.
- Aprovechar las métricas de prometheus para definir reglas que guíen las adaptaciones.

Bibliografía

[1] K. Birman, R. van Renesse, and W. Vogels, "Adding high availability and autonomic behavior to Web services," in *Proceedings. 26th International Conference on Software Engineering*, pp. 17–26, May 2004.

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

[3] 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.

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

[6] B. Foote and J. Yoder, "Big Ball of Mud," in *Fourth Conference on Patterns Languages of Programs*, (Monticello), Sept. 1997.

[7] R. C. Martin, "Chapter 15: What is an Architecture?," in *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, Robert C. Martin Series, London, England: Prentice Hall, 2018.

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

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] S. K. Mishra, B. Sahoo, and P. P. Parida, "Load balancing in cloud computing: A big picture," *Journal of King Saud University - Computer and Information Sciences*, vol. 32, pp. 149–158, Feb. 2020.

[16] J. Climent Penadés, "Disseny i prototipat de solucions autoadaptatives emprant arquitectures basades en microserveis. Una aplicació industrial pràctica," Master's thesis, Universitat Politècnica de València, Valencia, Oct. 2020.

[17] 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.

[18] J. Fons, "Especificación de sistemas auto-adaptativos," Mar. 2021.

[19] M. Gil, V. Pelechano, J. Fons, and M. Albert, "Designing the Human in the Loop of Self-Adaptive Systems," in *Ubiquitous Computing and Ambient Intelligence* (C. R. García, P. Caballero-Gil, M. Burmester, and A. Quesada-Arencibia, eds.), Lecture Notes in Computer Science, (Cham), pp. 437–449, Springer International Publishing, 2016.

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

[21] 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.

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

[23] 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.

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

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

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

[27] G. Roy, "Chapter 6. Message patterns via exchange routing," in *RabbitMQ in Depth*, Manning Publications, Sept. 2017.

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

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

[30] OpenAPI_Initiative, "OpenAPI Specification v3.1.0."

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

[32] 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.

[33] OpenTelemetry, "OpenTelemetry Documentation," 2022.

APÉNDICE A
APIs del Sistema

En este anexo incluimos la definición de todas las APIs de los microservicios del sistema. Esto incluye los *endponts* HTTP, las notificaciones y las peticiones asíncronas.

A.1 Monitorización

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

A.2 Conocimiento

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

A.3 Análisis

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

A.4 Planificador

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