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 Fons i Cors

Curso 2021-2022

Resum

????

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

Resumen

La Computación Autónoma (*Autonomic Computing*) promueve la ingeniería, diseño y desarrollo de sistemas con capacidades de autoadaptación, 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 autoadaptativos usando bucles de control MAPE-K.

Como resultado de este TFM se espera obtener la definición arquitectónica de soluciones autoadaptativas (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 listados de código

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. Se trata de las aplicaciones web.

Para estas, uno de sus requisitos clave 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 detectar y solucionar 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. Sería preferible 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 autoadaptativos**. Son aquellos capaces de ajustar su comportamiento en tiempo de ejecución en base a 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 preparar nuestros sistemas para adaptarse a entornos cambiantes, resolver conflictos operacionales e incluso a optimizarse dinámicamente.

Siguiendo con el ejemplo de la tienda on-line, un tipo de adaptación posible 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 autoadaptativas 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 VRAIN/UPV[4]). Esta propone una estrategia para la ingeniería de sistemas autoadaptativos usando bucles de control MAPE-K[2, 4].

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

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. Así se lograría 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. Rediseñar la arquitectura existente para soluciones autoadaptativas y prepararla 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: sondas, monitores, efectores...
3. Desarrollar un caso práctico para demostrar la viabilidad y aplicabilidad de nuestra propuesta.

1.3 Estructura de la memoria

El trabajo se divide en cuatro grandes secciones. La primera de ellas es el **marco teórico**. En el capítulo 2 se presentan algunos conceptos de la computación autónoma y los bucles de control. Se describirá la arquitectura MAPE-K, en la que se basa el trabajo. También definiremos 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, se introducirá el sistema actual: el bucle MAPE-K *Lite* de FaDA (capítulo 3). A continuación, en el capítulo 4 presentamos nuestra propuesta arquitectónica. Se describirá los distintos componentes que la conforman y los mecanismos de comunicación para conectarlos. Finalmente, en el capítulo 5, detallaremos paso a paso nuestra implementación.

La tercera sección del trabajo describe un **caso de estudio** (capítulo 6). En él, se desarrolla un sistema autoadaptativo básico de climatización. Nos sirvió para aplicar nuestra arquitectura en la práctica y verificar su funcionamiento. Sirve cómo implementación referencia para otras soluciones autoadaptativas.

Finalmente, cerraremos el trabajo con una descripción del proceso de **despliegue del sistema y pruebas** (capítulo 7). Para validar nuestra arquitectura realizamos distintos tipos de pruebas de funcionalidad y rendimiento. En base a los resultados, presentamos cambios que podrían hacerse sobre nuestra propuesta arquitectónica. A continuación, presentamos las conclusiones generales (capítulo 8). Se presentarán también las vertientes por las que se podría continuar ampliando el trabajo en un futuro.

Añadir diagrama de Gant con los 7 hitos

CAPÍTULO 2
Contexto Tecnológico

En este capítulo se presentan 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 se hará una breve introducción a las arquitecturas de *software*. Se detalla su motivación y los elementos que las componen. Es interesante comentarlas por dos motivos:

- En el trabajo se trata 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 sus 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.

Todos los sistemas *software* cuentan con una arquitectura. La diferencia radica en si 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": se implementan funcionalidades sin respetar la estructura. También se olvida el por qué de ciertas decisiones. Esto deriva en que se vuelva más difícil de mantener y desarrollar nuevas funcionalidades. Se convierte en una "gran bola de barro". [6]

Para evitarlo, es de vital importancia 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*». En definitiva, 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*». Es decir, pueden describirse 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 elemento con otros o con el entorno de operación. Describe ***cómo*** está organizado el sistema. Por ejemplo: un servicio A contacta con otro, B, usando una llamada HTTP.
- **Justificación**: Razonamiento o motivación de las decisiones que se han tomado. Responden al ***por qué*** algo se hace de una manera determinada. 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 interesaban especialmente los elementos. Concretamente los componentes y los conectores.

Componentes

El primer tipo de elemento a 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*.

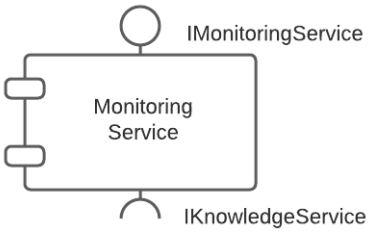


Figura 2.1: El servicio de monitorización representado como un componente. Ofrece una interfaz (*IMonitoringService*) y depende de otra para funcionar (*IKnowledgeService*).

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 e interacción entre ellos son aspectos clave que abordar.

Conectores

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

arquitectónicos que 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 librerías o *middlewares*. Sería el caso de un balanceador de carga que redirige peticiones a los nodos.

Una vez detectada la necesidad de comunicación entre dos componentes, es momento de evaluar qué mecanismo de comunicación es el más apropiado. Basándonos en nuestros requisitos, la arquitectura ya definida, y los mecanismos de despliegue que se quiera usar, elegiremos el conector oportuno. Podemos orientarnos con taxonomías como la de [10].

Fijémonos por ejemplo en la figura 2.2. En ella se muestran dos elementos que deben contactar. 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].

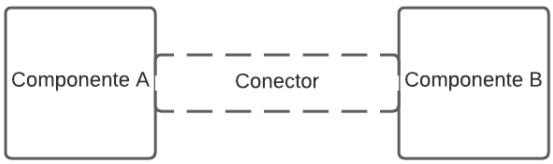


Figura 2.2: Ejemplo de comunicación de dos componentes a través de un conector.

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 permitirán 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 debe mantener o alcanzar 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 algunas 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, podría decidirse que es necesaria una acción correctiva. Por ejemplo, la acción podría consistir en desplegar nuevas instancias del servicio. Cuando su carga baje, podrá optar por eliminarlas.

Bucles de control

Para implementar las capacidades de adaptación se recurre a la teoría de control y al concepto de **bucle de control** (o *feedback loop*). [3] Se trata de un proceso iterativo para la gestión de sistemas. A partir de información sobre su estado 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):

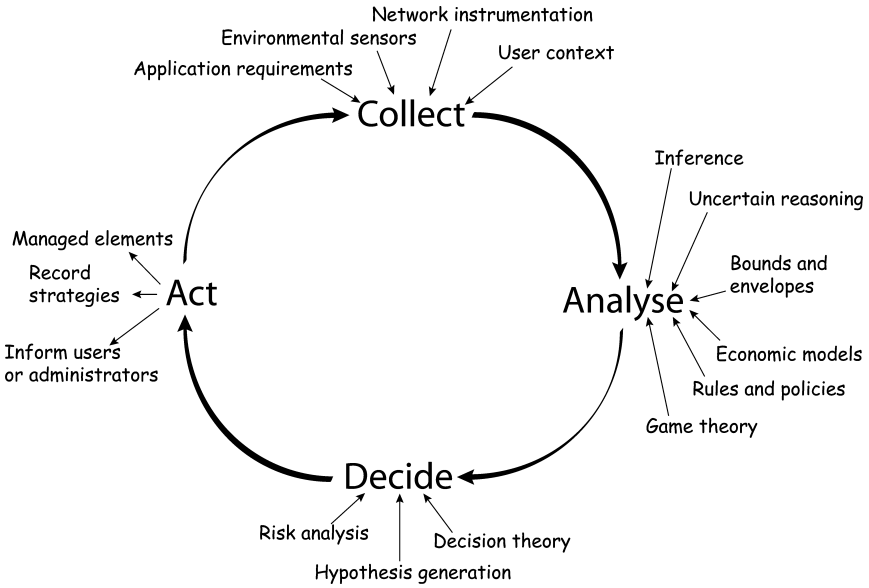


Figura 2.3: Un bucle de control genérico. Consta de cuatro actividades: Recopilar información, analizarla, decidir y actuar si procede. Obtenida de [13].

- **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 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. Podrían ser "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 se están 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 **ejecutarla** en esta etapa final. Mediante **efectores** en el sistema, el bucle es capaz de modificar su configuración. Dependiendo del éxito de la ejecución, la adaptación se completa o no. Finalizada esta etapa, se vuelve a recopilar información e inicia de nuevo el proceso.

En la ingeniería de *software*

En la ingeniería de *software*, los bucles de control suelen implementarse de dos formas distintas: **implícitos** o **explícitos**. La más habitual es la primera: se encuentran implícitos en el código de las aplicaciones. [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 acoplándose 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 de la red debe operar de forma autónoma y ser capaz de colaborar con el resto 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 mínima intervención humana.

Sus componentes principales son los **elementos autónomos**. Cada uno de los elementos del sistema es capaz de gestionarse y colaborar con el resto para alcanzar los

objetivos. Podría considerarse como una arquitectura basada en agentes. [17] A su vez, estos componentes pueden dividirse en dos partes: un recurso manejado y un manejador autónomo.

Los **recursos manejados** son las unidades de funcionalidad. Pueden ser de cualquier tipo, ya sea *hardware* o *software*. Para dotarlos de capacidad de autoadaptación, los emparejamos con un **manejador autónomo**: el bucle de control. Este gestiona al recurso en base a la información que recoge del entorno de ejecución y las políticas que guían su adaptación. Las **políticas** son los objetivos de alto nivel definidos por los administradores.

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).

2.3.1. Estructura del bucle MAPE-K

En la figura 2.4 mostramos una representación de un elemento autónomo. Pueden distinguirse las dos partes principales: el manejador y el recurso. El primero contacta con el recurso a través de sus sensores y efectores. Pueden apreciarse los componentes que conforman el bucle y que se describen a continuación: [2]

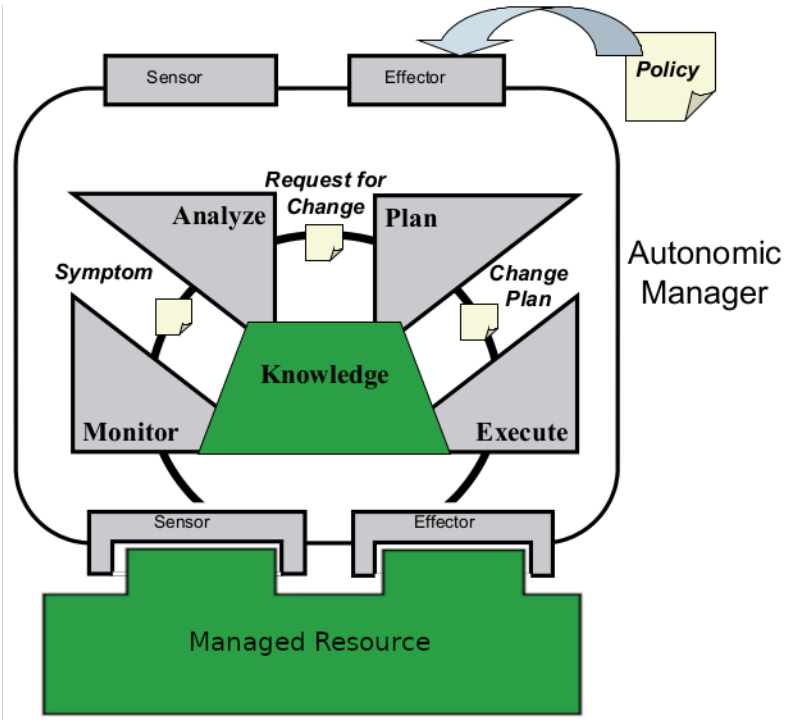


Figura 2.4: Representación de un elemento autónomo. Distinguimos el recurso manejado y el manejador autónomo. El manejador es un bucle MAPE-K. Basada en imagen de [2].

Para presentar estos componentes emplearemos 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 ser 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 el servicio web, una métrica relevante sería las 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. Esta se almacenará como propiedades de adaptación en la base de conocimiento.[18] El monitor y las sondas componen la etapa de recopilación de información de los bucles de control.

Siguiendo con el 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 el servicio.

Base de conocimiento

La base de conocimiento (*knowledge base*) es el componente base de 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 una **representación interna del estado del recurso manejado** que se mantiene en tiempo de ejecución. También se le conoce como el **modelo abstracto**. [12] Las demás etapas del bucle operan en base a él.

Analizador

La siguiente etapa del bucle corresponde con la de análisis. En base al modelo abstracto podemos razonar sobre su estado actual. Esto permite definir heurísticas para detectar situaciones que requieran de acciones correctivas, los **síntomas**. [3] Por ejemplo, que no se estén cumpliendo las políticas. Todo ello sin acoplarse al recurso manejado.

Continuando con el servicio web, definiremos heurísticas en base al número de peticiones por segundo. Podríamos definirlas como distintos valores umbrales que determinan si la carga del sistema es alta o baja.

Planificador

Si se ha detectado algún síntoma, el planificador describirá cómo se debe actuar. Comprobará el estado del sistema y pautará una serie de acciones correctivas. Estas se agruparán en un **plan de adaptación**. Las acciones correctivas se formulan en base a **operadores arquitectónicos**. [12] Dependiendo del estilo arquitectónico de nuestro sistema, tendremos disponibles una serie de operadores determinados. Estos suelen ser muy similares:

agregar o eliminar componentes, crear o eliminar conexiones entre ellos o modificar claves de configuración.

Respecto al servicio web, si la carga del sistema es muy alta, se podría planificar el despliegue de una nueva instancia. Cuando la carga baje, y si el servicio está replicado, podremos eliminarlas.

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 transmitiéndoles los comandos.

Si una adaptación se lleva a cabo correctamente, deberá reflejarse en el conocimiento. Para ello, una vez se confirme, las sondas reportarán su nuevo estado y seguirá el mismo proceso para almacenarlas como conocimiento. En cambio, 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 que este ofrezca (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. Cuando se complete, una sonda que reporte el número de instancias activas se lo comunicará al bucle.

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 indica que pueden actuar también como recursos manejados, reportando mediciones y ofreciendo efectores para manipularlo. 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 o en 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 elegir 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.

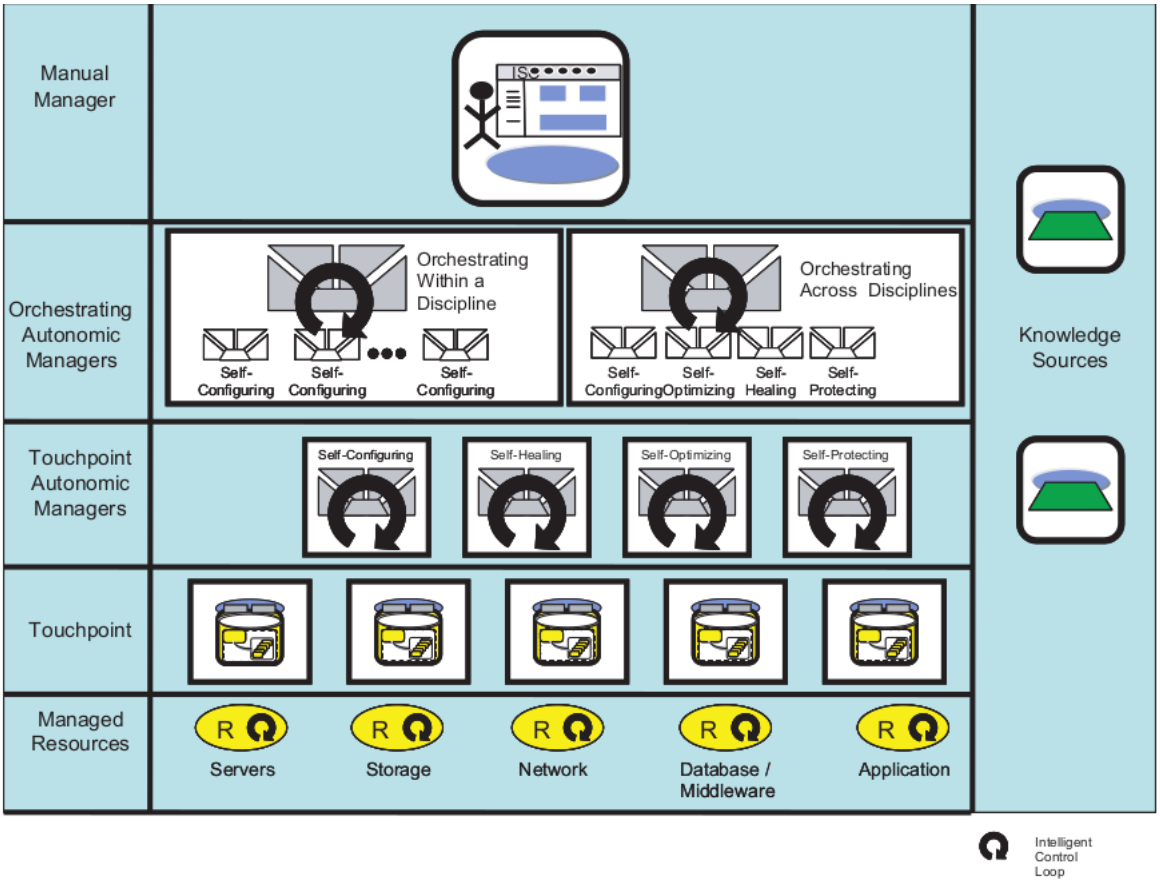


Figura 2.5: Arquitectura de un sistema autoadaptativo basado en MAPE-K. Imagen obtenida de [2].

CAPÍTULO 3

Sistema actual: plataforma FAdA y el bucle de control MAPE-K *Lite*

La plataforma FAdA[1] está enfocada en el desarrollo de sistemas autoadaptativos. Es un desarrollo del grupo PROS/Tatami[2] del instituto VRAIN/UPV[3]. Está compuesta por una serie de herramientas y guías metodológicas. Entre ellas encontramos extensiones de modelado para el IDE Eclipse, generadores de código y diversas implementaciones de bucles de control genéricos. Uno de estos bucles es aquel que queremos adaptar a entornos *cloud*: el bucle MAPE-K *Lite*.

En este capítulo haremos una breve introducción a la plataforma y las herramientas que la componen. Introduciremos conceptos como los servicios *adaptive ready* y describiremos el funcionamiento del bucle MAPE-K *Lite*.

3.1 Desarrollo de servicios *Adaptive Ready*

Comenzaremos describiendo la extensión para el IDE Eclipse. Tomando una aproximación de **desarrollo dirigido por modelos** (o *model driven development*, MDD), esta herramienta permite diseñar soluciones autoadaptativas. Las soluciones están compuestas por especificaciones de servicios ***adaptive ready*** (ARS). [4]

Los servicios *adaptive ready* implementan la funcionalidad y delegan en un bucle de control externo la responsabilidad de gestionar las adaptaciones. Para ello ofrecen una serie de interfaces, las sondas y efectores, que permiten recibir comandos desde el bucle. En la figura 3.1 se muestra un esquema del servicio y de cómo el bucle orquesta la solución.

La plataforma ofrece una serie de generadores de código. A partir de los modelos es posible generar todo el código de infraestructura de las interfaces de adaptación. Para integrarlas en los servicios, el desarrollador deberá importarlas e implementar la lógica asociada a los comandos. Dependiendo del tipo de despliegue por el que optemos, pueden generarse para microservicios, módulos OSGi[4], entre otros.

[1] Página oficial: http://fada.tatami.webs.upv.es/
[2] Página oficial: http://www.pros.webs.upv.es/
[3] Página oficial: https://vrain.upv.es/
[4] Página oficial: https://www.osgi.org.

Para operar estas soluciones, la plataforma ofrece distintas implementaciones de bucles de control. Cuál emplear dependerá del modelo de despliegue de la aplicación. Disponemos de soluciones para gestionar microservicios sobre la plataforma Kubernetes[4], otros que permiten operar componentes OSGi[5], etc. En este trabajo nos centraremos en una versión reducida del bucle de control para microservicios: MAPE-K *Lite*.

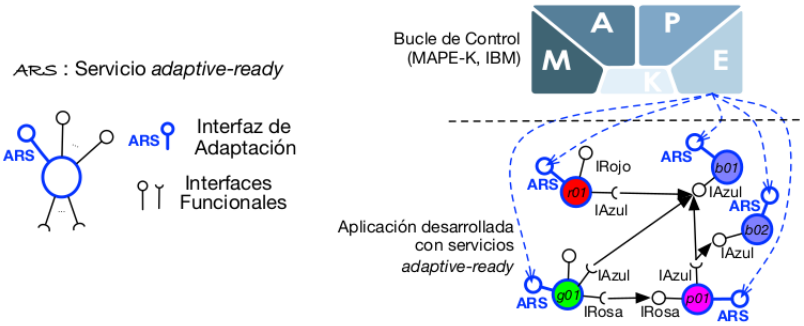


Figura 3.1: Servicio adaptive-ready (izquierda) y Bucle MAPE-K sobre arquitectura de microservicios adaptive-ready (derecha). Obtenida de [4].

3.2 Bucle de control MAPE-K *Lite*

El bucle de control MAPE-K *Lite* de FAdA nos permite gestionar soluciones autoadaptativas basadas en microservicios. Es una versión reducida del bucle presentado en [4]. Sigue la arquitectura MAPE-K descrita en la sección 2.3. En su implementación actual, se despliega como un servicio monolítico al nivel de la solución autoadaptativa. Todos sus componentes, tanto del bucle como los del recurso manejado (monitores, reglas...), se ejecutan dentro del mismo proceso.

El bucle de control es genérico. No está acoplado a un dominio o solución concreta. Además, pueden hacer uso de él varios sistemas a la vez. El proceso acepta como entradas las mediciones de las sondas y emite comandos dirigidos a los efectores del recurso manejado. En nuestro caso, los *touchpoints* de los servicios *adaptive ready*. En la figura 3.2 mostramos un modelo que detalla el flujo de control e información dentro de sus etapas.

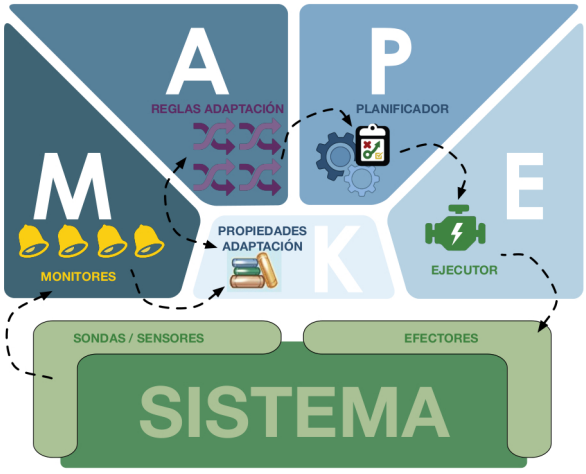


Figura 3.2: Arquitectura del bucle MAPE-K *lite*. El flujo de información y de control entre las etapas del bucle están representados con flechas. Obtenida de [18]

[5]Página oficial: https://www.osgi.org.

3.2.1. Estructura del bucle

En la sección 2.3.1 ya describimos la estructura de un bucle MAPE-K típico. Como se puede ver en la figura 3.2, este no difiere mucho en su implementación. Sigue la misma secuencia de monitorización, análisis, planificación y ejecución. Para implementarlas cuenta también con los mismos componentes: sondas, monitores, módulo de análisis, planificador, ejecutor y efectores. Pero sí que hay una faceta en la que queremos hacer hincapié: el uso de reglas de adaptación en la etapa de análisis.

Módulo de análisis y reglas de adaptación

El módulo de análisis del bucle de control se ha implementado mediante un conjunto de **reglas de adaptación**. Se trata de la codificación de heurísticas que nos permiten corregir u optimizar el funcionamiento del sistema. En base a los reportes de los monitores, se buscan oportunidades para mejorar la configuración en tiempo de ejecución.

Las reglas pueden ser generales o especificas a un recurso manejado concreto. Se dividen en dos componentes: la condición y la propuesta de cambio. La **condición** es una función que determina si es necesario ejecutar la acción. Se define a partir de las propiedades de adaptación del conocimiento. Esta se evaluará en tiempo de ejecución cuando se produzca un cambio en la configuración o se obtenga nueva información del entorno.

Por otro lado, la **propuesta de cambio** describe cuál debería ser la siguiente configuración del sistema: qué componentes deben estar o no presentes, qué conexiones entre ellos deben existir y sus parámetros de configuración. Las reglas enviarán esta propuesta al planificador. El planificador analizará el estado actual y pautará qué **acciones** son necesarias para alcanzar el estado deseado. Sobre los componentes y conexiones que no se especifique nada se mantendrán como estaban.

Operadores arquitectónicos

Las acciones se especifican usando **operadores arquitectónicos**. [12] Dependiendo del estilo arquitectónico del recurso manejado, tendremos disponibles unos operadores determinados. El bucle de control que nos ocupa es específico para gestionar soluciones basadas en microservicios. Por tanto, ofrecerá los operadores correspondientes a este estilo. En [18] podemos encontrar los cinco tipos que ofrece:

- **Desplegar servicios** (***deploy***): Añadimos una nueva instancia de un servicio.
- **Eliminar servicios** (***undeploy***): Eliminamos una instancia de un servicio.
- **Enlazar servicios** (***bind***): Añadimos una conexión entre dos servicios. A partir de entonces podrán comunicarse.
- **Desenlazar servicios** (***unbind***): Eliminamos una conexión existente entre dos servicios. Ya no podrán comunicarse.
- **Cambiar configuración** (***set parameter***): Modificamos un parámetro de configuración de un servicio.

En la figura 3.3 mostramos un ejemplo de los efectos de estas acciones en la arquitectura del sistema.

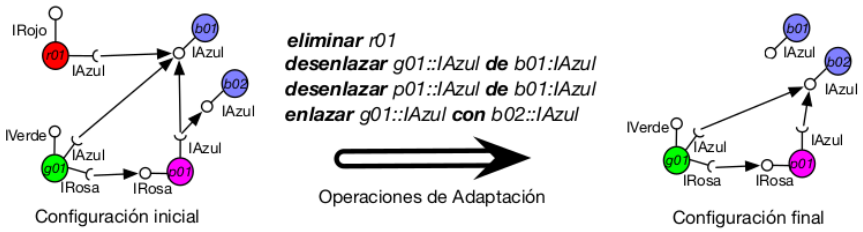


Figura 3.3: Ejemplo de adaptaciones en un sistema basado en microservicios. Imagen original obtenida de [4]

Ejemplo de regla de adaptación

Para presentar una regla de adaptación, nos adelantaremos un poco y tomaremos de ejemplo una de las definidas en el caso de estudio (capítulo 6). Esta regla pertenece a un sistema de climatización encargado de regular la temperatura de una habitación. Este cuenta con un termómetro (la sonda) y un sistema de aire acondicionado (el recurso manejado). Además, el recurso cuenta con una serie de efectores que permiten cambiar el modo de funcionamiento: enfriar, calentar o apagado.

Esta regla en concreto se encarga de activar el modo refrigeración. Para ello, en su condición comprueba que la temperatura supere un umbral definido por el usuario; y que no esté enfriando ya la estancia. De cumplirse, se especifica que en la siguiente configuración del sistema deberá estar presente el componente de aire acondicionado. Este debe deberá tener activo el modo de enfriamiento.

Para especificar las reglas usaremos la notación SAS. [18] Esta define una serie de pautas para describir su condición y representar la configuración que solicitará el cuerpo de la regla. En la tabla 3.1 presentamos la especificación de esta regla.

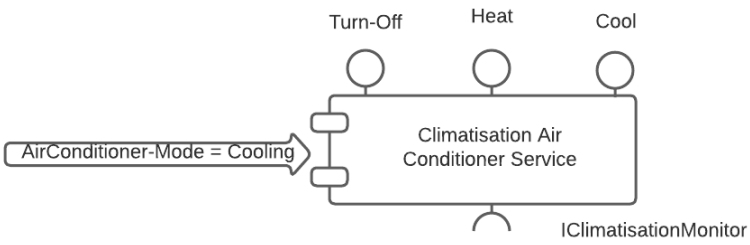
Regla:	`EnableAirConditionerCoolingModeWhenTemperatureThresholdExceeded`
Descripción:	Activa el aire acondicionado en modo enfriar cuando la temperatura sea superior al umbral de calor.
Condición:	*airconditioner-mode* != *Cooling* **AND** *temperature* >= *hot-temperature-threshold*
Cuerpo:	

Tabla 3.1: Ejemplo de especificación de regla de adaptación con la notación SAS.

El planificador del sistema recibirá esta propuesta de cambio y tendrá que determinar qué acciones son necesarias para alcanzarlo. Por ejemplo, si el componente del aire acondicionado estuviera activo, sólo se pautaría la acción para cambiar el parámetro `Mode` para que tenga el valor `Cooling`.

CAPÍTULO 4
Diseño de la solución

En este capítulo describiremos la estrategia que seguimos para refactorizar el bucle MAPE-K *Lite*. Como comentamos en la introducción, nuestro objetivo era adaptarlo a entornos *cloud*. Esto implica transformarlo de un servicio monolítico a un sistema distribuido basado en microservicios. Se trata de un cambio arquitectónico importante. Por ello, queríamos seguir una estrategia ingenieril que tenga en cuenta las particularidades del sistema.

Comenzaremos justificando la elección de este estilo arquitectónico y describiendo los beneficios que esperábamos obtener. A continuación, detallaremos nuestro diseño y explicaremos cómo se dividió la funcionalidad del bucle en microservicios. Después, presentaremos algunos de los mecanismos de comunicación más populares y comentaremos cuáles elegimos para conectar nuestros componentes. Finalmente, presentaremos la estructura final de nuestra arquitectura.

4.1 ¿Por qué usar microservicios?

Una de las decisiones clave que tomamos durante el desarrollo de este trabajo fue elegir el estilo arquitectónico a emplear. Dado que queremos adaptar el bucle de control a entornos en la nube, será imprescindible que esté preparado para ellos. Optamos por las **arquitecturas basadas en microservicios** porque son uno de los pilares de las aplicaciones *cloud native*. [20] Gracias a este estilo, obteníamos una serie de beneficios que consideramos vitales para extender el bucle MAPE-K.

El más evidente fue poder **independizar la implementación** de los servicios. Si fuera necesario, podríamos emplear tecnologías distintas para cada uno. Distintas plataformas o lenguajes de programación que nos permitan ofrecer la funcionalidad requerida. Incluso podríamos contar con implementaciones alternativas de algunos componentes. Estos podrían ofrecer diferentes estrategias de la misma funcionalidad: distintos planificadores, módulos de análisis, etc.

También vimos muy importante el desarrollar **servicios *plug & play***. Al añadir o eliminar componentes podríamos cambiar las capacidades de adaptación del sistema en tiempo de ejecución. Sin necesidad de pararlo en ningún momento. Por ejemplo, al desplegar un nuevo conjunto de reglas de adaptación se dotaría al bucle de capacidades para gestionar otros aspectos de los recursos.

Por otro lado, facilitaría **escalar la capacidad computacional** de cada servicio de forma independiente. Si uno en concreto estuviera recibiendo más peticiones que los demás, podemos limitarnos a desplegar una nueva instancia de este componente. No sería ne-

cesario desplegar el bucle completo. Esto nos permitirá aprovechar mejor los recursos disponibles.

Finalmente, nos ayudaría a desacoplar el bucle de los componentes específicos para gestionar una solución autoadaptativa determinada. Así, en un futuro, podríamos aprovechar la misma infraestructura para manejar varias soluciones simultáneamente. Esto se conoce como multicliente o ***multi-tennancy***. Como tiene importantes implicaciones a nivel de seguridad y gestión de los datos de los distintos clientes [21], en este trabajo no profundizaremos más allá de separar los componentes.

Optar por un estilo arquitectónico determinado tiene un precio. Conlleva una serie de desventajas que no podremos ignorar. Aun así, se consideró que los beneficios las compensaban. Entre ellas, la más importante es un **aumento en la complejidad** del sistema. [22] Pasamos de trabajar con un solo servicio monolítico a tener que gestionar varios más pequeños.

A nivel de diseño, esto afectará a la **comunicación entre servicios**. Como ya no se ejecutan dentro del mismo proceso, deberán comunicarse a través de la red. Más adelante veremos que esta no es infalible. [23] Pueden ocurrir retrasos para entregar los mensajes o directamente fallar la conexión. Por tanto, necesitaremos plantear bien cómo y cuándo deberán contactar entre ellos. Lo ideal es que sean lo más independientes posible, ya que cada dependencia es un potencial punto de fallo.

Otros aspectos que se verán afectados serán la **operación y el despliegue del sistema**. Para su correcto funcionamiento tendremos que monitorizar y gestionar cada una de sus piezas. Como una sola petición puede desencadenar llamadas a otros servicios, requeriremos de soluciones de monitorización y *logging* más avanzadas. [24] Sin una buena solución de **observabilidad** puede ser muy difícil de depurar los errores o analizar su impacto.

4.2 Distribución de los componentes

Una vez elegido el estilo arquitectónico, el siguiente paso fue identificar qué servicios compondrían nuestra arquitectura. Por suerte, partíamos de un sistema existente bien documentado. Conocíamos el rol de cada uno de sus elementos y sus requisitos. Nos quedaba entonces determinar cómo agrupar estos elementos en servicios. ¿Cómo definimos las fronteras entre cada uno de ellos? ¿Qué componentes debe abarcar cada microservicio?

Para ello nos guiamos por unas pautas para dividir aplicaciones monolíticas. [22] Los servicios que definamos deben presentar **alta cohesión** en su funcionalidad. Es decir, siempre que sea posible debemos agrupar los componentes relacionados en un mismo servicio. De lo contrario, acabarían muy acoplados entre sí. No serían realmente independientes y necesitarían comunicarse constantemente. En caso de que uno de ellos falle, el resto no podría operar con normalidad.

Por este motivo, una de las primeras decisiones que tomamos fue **separar cada etapa del bucle a su propio servicio**. Entre ellas existe una división funcional bien definida y pueden operar con cierta independencia. Además, como el conocimiento es compartido entre todas las etapas, también tendría su propio microservicio. Esto nos reportaría los beneficios que comentamos en la sección anterior como las implementaciones independientes y la posibilidad de escalar su capacidad de cómputo individualmente.

La siguiente división que consideramos fue **separar del bucle de control los componentes específicos de las soluciones manejadas**. Se trata de los monitores, reglas y demás elementos que conocen el dominio de la solución. Actualmente se despliegan en

el mismo proceso que el bucle. En caso de querer cambiar alguno, tendríamos que redesplegarlo entero. Extrayéndolos a servicios independientes, el bucle se volverá agnóstico a las soluciones manejadas. En tiempo de ejecución podremos añadir o eliminar estos componentes, sin que afecte a su funcionamiento.

En la figura 4.1 mostramos la arquitectura resultante de estas divisiones de funcionalidad. **Cada elemento es un tipo distinto de microservicio**. Las sondas y los efectores son un caso especial. Podrán desarrollarse como microservicios o como elementos empotrados en el recurso manejado. Dependerá de si tenemos control o no sobre la implementación del recurso.

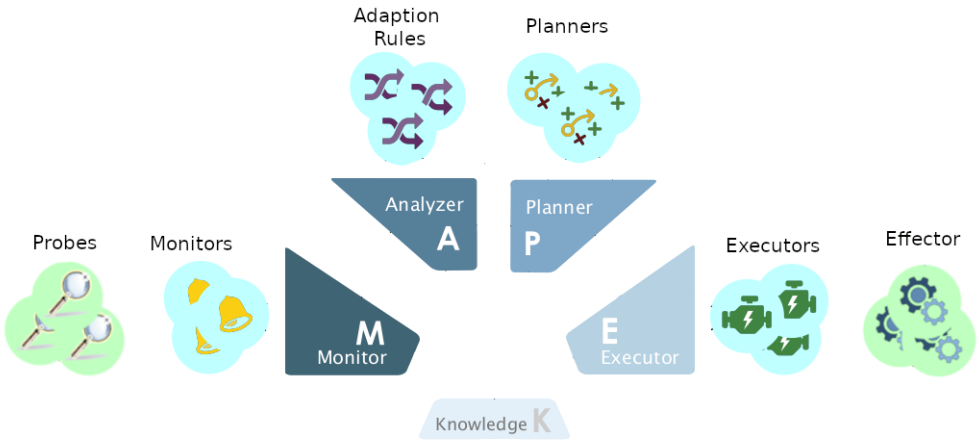


Figura 4.1: Diagrama con los microservicios que conforman nuestra arquitectura distribuida.

4.3 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*.

Para elegir los mecanismos de comunicación, investigamos algunos de los estilos arquitectónicos existentes. Finalmente, nos decantamos por las **arquitecturas de servicios jerarquizados**. Estas nos permitían explotar la separación entre el bucle de control y los recursos manejados. En concreto, nos inspiramos en el estilo arquitectónico C2 (*components and connectors*)[11, 25].

4.3.1. Jerarquías de microservicios: Arquitectura C2 y arquitectura limpia

Este estilo organiza sus elementos en jerarquías o capas. Un componente se ubicará en un nivel determinado en base a su nivel de abstracción respecto al entorno de ejecución. En las capas inferiores se encuentran aquellos más externos, los más "acoplados al mundo real". Por ejemplo, las interfaces de usuario estarían en esta capa. Por otro lado, en las capas superiores se encuentran los servicios con mayor abstracción. Este sería el caso de las reglas de negocio de un programa. No hay ninguna restricción en cuanto al número de capas, podemos contar con cuantas sea necesario.

En cuanto a la comunicación, un componente solo puede contactar con sus vecinos inmediatos (en una capa superior o inferior). Limitando su contacto con otras capas limi-

tamos su alcance y su conocimiento sobre el resto del sistema. Así evitamos que se acople al resto de componentes más de lo necesario. Además, dentro de un mismo nivel no pueden contactar entre ellos. Según la dirección de la comunicación, se emplean mecanismos distintos (figura 4.2):

- **Peticiones** (*requests*): Se trata de solicitudes a un servicio concreto para que ejecute una acción. Un componente se comunica 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 usuario a la interfaz gráfica entraría en esta categoría.
- **Notificaciones**: Un componente envía un mensaje hacia abajo en la jerarquía, sin especificar un receptor. Todos los servicios que estén por debajo lo recibirán y decidirán si tratarlo o no. Esto evita que nuestro componente se acople a ellos. Se puede emplear para comunicar eventos que puedan ser de interés al resto. Un ejemplo sería notificar al resto de servicios sobre la creación de un nuevo usuario.

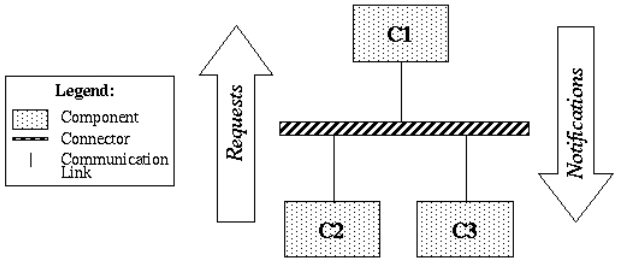


Figura 4.2: Ejemplo del estilo arquitectónico C2 (*Components and Connectors*). [25]

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. También se encuentran los elementos que nos permiten interactuar con él: las sondas y efectores. Hacen de intermediarios entre el recurso y el resto del bucle.
- **Nivel de solución**: En esta capa se encuentran los componentes del bucle específicos para una solución autoadaptativa: monitores, reglas de adaptación, etc. No los incluimos en el mismo nivel que las sondas y efectores porque deben comunicarse con ellas. Estos elementos mantienen una dependencia con el propio bucle de control.
- **Nivel del bucle**: Aquí se encuentran los servicios de las etapas del bucle: 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, limitando su acceso. Por ejemplo, ningún servicio salvo los monitores deberían poder modificarlo.
- **Conocimiento**: Es la capa más interna y la base de nuestra 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*. [26] 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 muestra dependencia sólo con la capa a la que envuelve. Sólo puede comunicarse con componentes dentro de esta.

Basándonos en la descripción anterior, optamos por representar nuestra arquitectura como *knowledge driven*. [5] Nuestra capa central será la del conocimiento (amarilla). A partir de ahí, cada nivel superior dependería de aquella a la que envuelve: el bucle al conocimiento (rojo), la solución al bucle (verde), y la del recurso manejado a la solución (azul). En la figura 4.3 mostramos el resultado. Las flechas negras representan las peticiones; y las moradas, las notificaciones.

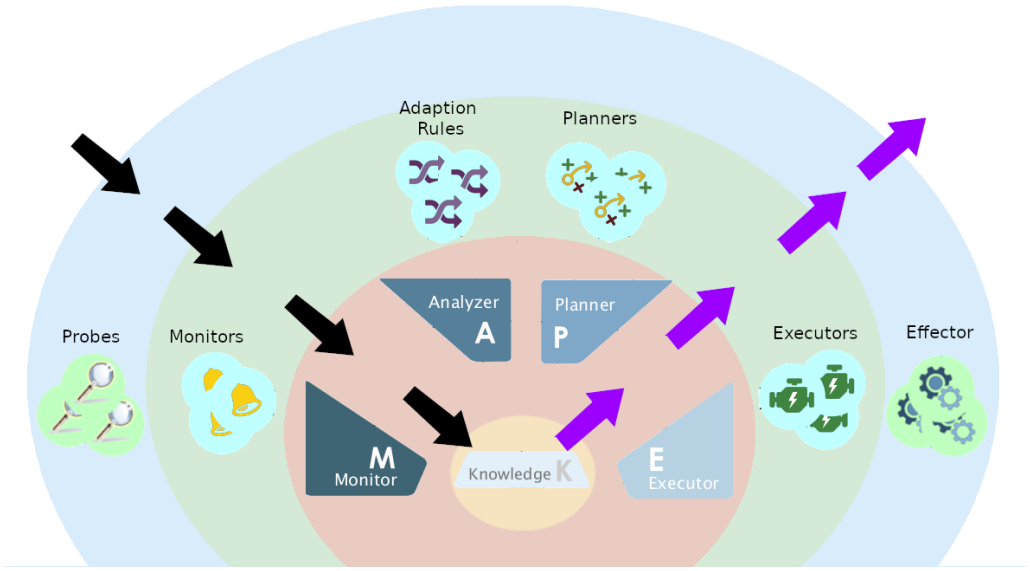


Figura 4.3: Representación de nuestra propuesta arquitectónica. Inspirado en Arquitectura Limpia (*Clean Architecture*). [1]

4.3.2. 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 de 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. El problema estaba en que ambos componentes se encuentran en la misma capa. Como dependen del conocimiento para funcionar, no podíamos moverlos a una superior para emplear las notificaciones.

Requeríamos entonces de un tercer patrón de comunicación. Detectamos que este debería de ser dirigido: los mensajes van destinados a un tipo de componente específico. Y, 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

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

patrones existentes. Lo presentamos a continuación 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.

4.3.3. Conectores

Una vez determinados los mecanismos de comunicación, comenzamos la búsqueda de los conectores para implementarlos. Seguimos entonces la estrategia descrita en [5]. Esta consiste en investigar las necesidades de comunicación entre componentes y, a partir de sus requisitos, determinar el **tipo de conector adecuado**. Sabiendo que hemos optado por una arquitectura distribuida, la elección se simplifica: los servicios pueden estar desplegados en máquinas distintas, por lo que el paso de mensajes será a través de la red.

Debido a esto, no recurrimos a la taxonomía que lista [10]. En su lugar 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.

Todas estas tecnologías siguen el principio de ***smart endpoints and dumb pipes*** (servicios inteligentes y conductos tontos) [27]. Se trata de mecanismos de comunicación ligeros, como HTTP, sin mucha lógica asociada para regularla. Se limitan a ser simples conductos que transmiten los mensajes entre servicios. En su lugar, será el emisor será el responsable de gestionarla. Por ejemplo, implementando políticas de reintento o gestionando el enrutamiento de mensajes.

Para elegir los conectores evaluamos las tecnologías mediante un análisis de *trade-offs*. A continuación, detallamos las ventajas y desventajas que detectamos de cada uno.

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. [22] 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**. [28] Este permite al programa cliente interactuar con objetos que se encuentran en servidores remotos como si se encontraran localmente. Se

implementa mediante objetos que actúan como *proxies*, abstrayendo al cliente de la llamada al servidor.

Los *proxies* ofrecen una interfaz para que el cliente invoque sus métodos localmente. Por debajo, estos realizan una llamada al servicio remoto donde se encuentra realmente. El servidor remoto procesa la petición y nos devolverá un resultado. En la figura 4.4 incluimos un diagrama 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.

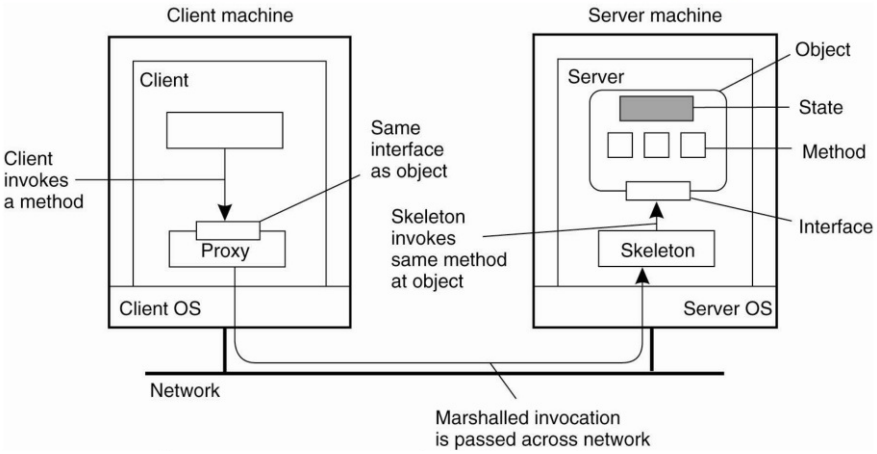


Figura 4.4: Funcionamiento del sistema de objetos distribuidos. Obtenido de [28].

- **Ventajas**:
  - El servidor puede ofrecer todo tipo de operaciones a sus clientes. No estamos limitados por un estándar.
  - Permite distribuir la carga de procesamiento del sistema. Esto puede ayudar para escalar la aplicación.
  - **Comunicación síncrona**: Es un mecanismo ideal para comunicaciones síncronas. En ellas, el cliente requiere la respuesta del servicio para poder continuar con su procesamiento.
  - **Comunicación asíncrona**: Si el cliente no requiere de una respuesta del servidor, puede enviar el mensaje y continuar con su ejecución. Esta estrategia es conocida como *fire and forget* (disparar y olvidar).
  - (*Objetos distribuidos*) Se abstrae al cliente de la interacción con un servidor remoto. Esto facilita su implementación. Al estar tipado, es más sencillo construir las peticiones.
- **Desventajas**:
  - **Dirigida**: Necesitamos conocer de antemano la ubicación del servidor al que queremos hacer una petición.
  - Dificulta la integración con otras aplicaciones. Cada servicio ofrecerá sus propias funciones. No están estandarizadas.
  - (*Objetos distribuidos*) Tendremos que actualizar el cliente con cada cambio en el esquema del servidor.

- (*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. [23]
- (*Objetos distribuidos*) Sistemas como Java RMI son específicos para una plataforma concreta. n este caso, Java. Perderemos flexibilidad en cuanto a qué otras tecnologías podemos emplear en nuestra arquitectura. [22]

***Representational State Transfer* (REST)**: Este mecanismo está basado en RPC, pero con ciertas restricciones adicionales. [5] Sigue también el modelo cliente-servidor. Su concepto principal son los **recursos**: cualquier elemento sobre el que el servicio pueda ofrecernos información. Estos deben tener asociados un identificador único (una URI). [29] Ejemplos de recursos podrían ser las entidades del dominio que gestiona nuestro servicio: usuarios, publicaciones...

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 ofrecen es **uniforme**. Solo cambia el "esquema de los datos", los tipos de recursos que sirven a los clientes. Esto facilita enormemente la integración con otros servicios. [30] La implementación más habitual es sobre el protocolo HTTP (*Hypertext Transfer Protocol*). Este define métodos estandarizados como `GET` para las lecturas, `PUT` para las actualizaciones, etc.

Otra de las pautas que dicta el estilo es que el servidor no puede mantener el estado de las sesiones de la aplicación (deben ser ***stateless***). Esto significa que cada petición debe poder procesarse de forma independiente. Esto permitirá por un lado que el servidor sea más eficiente, ya que no necesita recordar todos los datos de las sesiones en curso. Por otro lado, si nuestro servicio está replicado, cualquier instancia podrá atender la petición.

- **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. Las distintas instancias puedan atender las peticiones que surjan durante una misma sesión.
  - **API Uniforme**: Los métodos que exponen estos servicios están estandarizados y son sencillos. Un servidor solo debe implementar unos pocos métodos estándar que consumirán los clientes. Esto mejora significativamente la **interoperabilidad**.
  - **Comunicación síncrona**: Es un mecanismo ideal para el patrón *request-response*. En ellas, el cliente requiere la respuesta del servicio para poder continuar con su procesamiento.
  - **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 4.3.4 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 Uniforme**: 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 el estilo REST).

GraphQL[2]: Es protocolo de consultas para APIs. Sigue también el estilo cliente - servidor. El servidor expone un *endpoint* que permite consultar los datos de la aplicación en forma de **grafo**. Cada **nodo** tiene asociado un **esquema de datos**. Este describe sus propiedades y relaciones con otros. Partiendo de un esquema determinado, podemos realizar consultas que naveguen este grafo. También permite el uso de filtros.

Dota de enorme **flexibilidad** a los clientes. Con una sola consulta son capaces de recuperar toda la información que consideren relevante. Se evita el ***over-fetching***, recuperar datos irrelevantes. [31] Esto ha provocado que empiece a ganar popularidad frente al estilo REST para realizar consultas complejas. [32] En REST, estamos limitados a los esquemas de datos de las vistas que ofrezca un recurso. No podemos modificarlos mediante consultas. Incluso, es posible que requiramos de varias llamadas distintas a la API para construir la misma respuesta que en GraphQL.

- **Ventajas**:
  - **Consultas más expresivas**: Nos permite definir consultas tan complejas como necesitemos.
  - **Rendimiento**: Es ideal para entornos donde queremos optimizar el uso de red. Esto es gracias a que se reduce la cantidad de llamadas a la API y podemos obtener solo la información relevante.
    - Esto lo hace **ideal para móviles**.
- **Desventajas**:
  - **Solo permite lecturas**: Es un lenguaje de consultas. No tiene comandos que permitan escrituras.
  - **Problemas de rendimiento**: Los clientes pueden 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. En él, contamos con un servicio que actúa como intermediario de la comunicación, el ***broker. [22] Este nos permite **desacoplar** a los participantes. [33] El emisor de un mensaje no necesita conocer detalles de los destinatarios: su dirección, el número de instancias, si están activos en este momento, etc. Sólo necesita conocer el formato del mensaje, las colas o temas asociados; y enviárselo al *broker*. Este se encargará de enrutarlo.

Uno de los conceptos más característicos de este estilo son las **colas de mensajería**. Se trata de colecciones FIFO (*first in, first out*) donde el *broker* almacena los mensajes destinados a un servicio **consumidor**. Cada vez que se añada un mensaje nuevo, el consumidor será notificado. Este podrá ir procesándolos secuencialmente, sin necesidad de saturarse.

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

Las colas nos permiten implementar varias estrategias de comunicación: colas de trabajo, *publish-suscribe*, híbrida...

Tomemos por ejemplo las **colas de trabajo**. [34] Esta estrategia nos permite implementar comunicaciones asíncronas punto a punto (dirigidas). Es ideal para casos en los que necesitamos que un mensaje llegue sólo una vez a un destinatario determinado. [35] Por ejemplo, para enviar peticiones de trabajo que pueden ser costosas de procesar. Para enviarlas, el emisor sólo necesita conocer el nombre de la cola de mensajería correspondiente y usar el formato de mensaje apropiado. No requiere conocer ningún otro detalle. En la figura 4.5 mostramos un ejemplo con un productor (P) que envía el mensaje a una cola con dos consumidores (C1 y C2) a la escucha de esta.

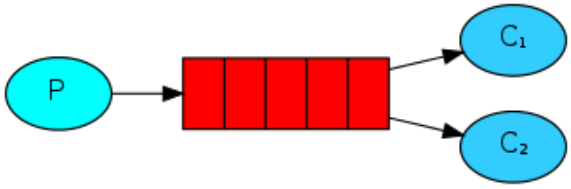


Figura 4.5: Representación de las colas de trabajo. Ejemplo de comunicación asíncrona dirigida. [3]

Otra estrategia que podemos implementar es ***publish-suscribe***. Esta sirve para implementar comunicación *broadcast*. [34] Se basa en el uso de temas o ***topics***, categorías de mensajes. En base a ellas, los servicios consumidores pueden suscribirse a las que le resulten de interés. Un servicio (el productor) enviará un mensaje al *broker*, indicando que pertenece a un tema determinado. El *broker* recibe el mensaje y se encarga de enrutarlo a las colas de todos los consumidores suscritos a él. [36] En la figura 4.6 tenemos un ejemplo. El mensaje "A" llega a todos los servicios suscritos a al tema "Topic".

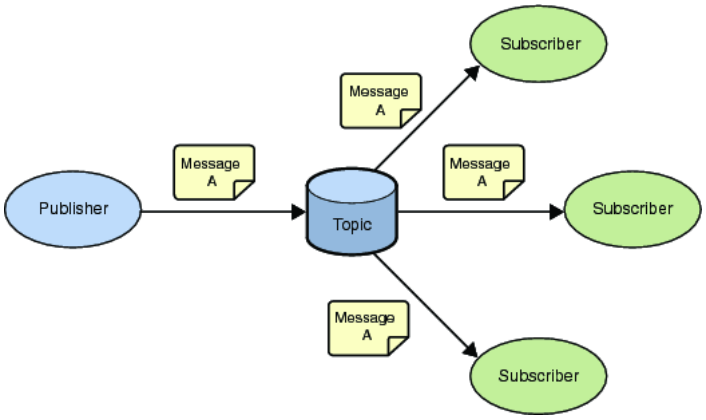


Figura 4.6: Estrategia *publish/suscribe*: el *broker* actúa como intermediario en la comunicación *broadcast*. Imagen obtenida de [4].

- **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. Por otro lado, el consumidor puede ir procesándolas a su ritmo.

[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

- **Escalable**: Varias instancias del mismo servicio pueden estar a la escucha de la misma cola. De esta forma, podemos aumentar la capacidad de cómputo de forma transparente a los emisores.
- **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*.
- **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]
  - **Desacoplamiento de componentes**: Debido al alto nivel de desacoplamiento, puede resultar difícil detectar cuando el consumidor no está recibiendo los mensajes. Por ejemplo, si el emisor los está publicando en una cola equivocada. Los mensajes se acumularían en esta cola errónea y el consumidor real no recibiría nada.
  - **Envío garantizado de mensajes**: Si se reintenta el envío del mensaje, es posible que ya lo hayamos consumido. 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 y cardinalidad	Dirigida (1..1)	Dirigida (1..1)	Dirigida (1..1)	Dirigida (1..1) y *broadcast* (1..N)
Acoplamiento entre componentes	Alto	Medio	Alto	Bajo
Interoperabilidad	Baja	Alta	Alta	Alta[5]
Peticiones de lectura	Sí	Sí	Sí	Sí[6]
Peticiones 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.

4.3.4. Peticiones síncronas

Comenzamos investigando el diseño de las peticiones síncronas. Tomemos por ejemplo la comunicación entre el servicio de monitorización (*monitoring service*) y el servicio de conocimiento (*knowledge service*). Recordemos que el conocimiento almacena todas las

[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.

propiedades de adaptación. El resto de los servicios del nivel del bucle necesitan consultarlas y actualizarlas durante su funcionamiento. En la figura 4.7 representamos inicialmente ambos componentes y un conector, sin especificar de qué tipo será.

El siguiente paso consiste en identificar qué interacciones debe existir entre ambos componentes. En este caso, el servicio de monitorización debe poder leer y actualizar el valor de las propiedades. Por tanto, existen operaciones de lectura y escritura de los datos.

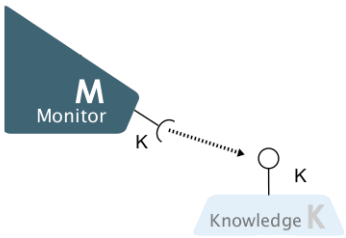


Figura 4.7: Representación inicial del conector entre el servicio de monitorización y el de conocimiento. De momento no indica más que la necesidad de comunicación.

De entre los cuatro tipos de conectores, pudimos descartar inmediatamente la opción de GraphQL. Se trata de un mecanismo más orientado a las consultas de datos. En nuestro caso, necesitamos ejecutar también escrituras. Aunque podríamos trabajar con dos protocolos de comunicación en paralelo, esto aumentaría la complejidad de la arquitectura. Además, el protocolo está más enfocado en consultas complejas. Como las propiedades se almacenan como pares clave - valor, sin relaciones, no lo consideramos necesario.

También descartamos emplear el *broker* de mensajería. Como requerimos de lecturas de datos, nos convenía más recurrir a otros patrones. Para obtener propiedades del conocimiento, resultaba más sencillo de implementar mediante comunicación síncrona.

Finalmente, teníamos que decidir entre usar REST o RPC. Para tomar la decisión decidimos priorizar la **interoperabilidad**. Este componente estará expuesto "hacia fuera": expondrá su interfaz a la capa superior. Cualquier elemento que se encuentre en ella podrá contactarlo. Potencialmente, estos podrían ser desarrollados por terceros. Para maximizar la compatibilidad, descartamos entonces RPC. Este protocolo nos hubiera acoplado a una tecnología concreta y a APIs no estándares.

Nos terminamos decantando entonces por el conector REST. Implementamos ambas funciones de lectura y escritura mediante *endpoints* HTTP. Su especificación se detalla a continuación en las tablas 4.2 y 4.3. En ellas se incluye la operación y la ruta sobre que aplica.

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 concuerda con el 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 concuerda con el 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 REST. Nos permite especificar 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.[37] 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. Especifica las operaciones HTTP que se pueden ejecutar sobre estos recursos y las estructuras de datos de entrada o salida. También se incluyen los códigos de respuesta de HTTP: códigos indican al cliente el resultado de la ejecución de la operación. [37] Más adelante, en el fragmento 5.2, mostraremos un ejemplo de un documento típico.

La especificación puede escribirse manualmente o puede generarse a partir de una implementación existente. Así, podemos desarrollar primero nuestro servicio en un determinado lenguaje y obtener después su descripción en OpenAPI. Además, se podrá 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. [38]

Uno de los casos de uso más interesantes es la **generación de código**. 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 ya descrito en el apartado de RPC.

Para el desarrollo de este trabajo, nos interesaba especialmente debido a las posibles diferencias tecnológicas entre microservicios. Cada uno de ellos puede implementarse utilizando distintos lenguajes o librerías dependiendo de la funcionalidad a ofrecer. Gracias a la generación de código, se podría obtener la especificación del servidor y generar los clientes o servidores en cualquiera de los lenguajes soportados.

Arquitectura del conector

Para terminar, mostramos la estructura del conector que empleamos para implementar las peticiones síncronas. Aparece en la figura 4.8. Esta muestra como el servicio de monitorización contacta al de conocimiento para asignarle un valor a la propiedad *Temp* (temperatura).

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

El conector, delimitado por una línea discontinua roja, está compuesto por dos elementos: una API REST y un cliente. Los otros dos grupos representan los procesos de los servicios de monitorización y conocimiento. El servicio de monitorización se comunica con la API través del API Client. Este se despliega dentro de su proceso actuando como *proxy*. En el capítulo 5 - Implementación describiremos nuestra propuesta para su implementación.

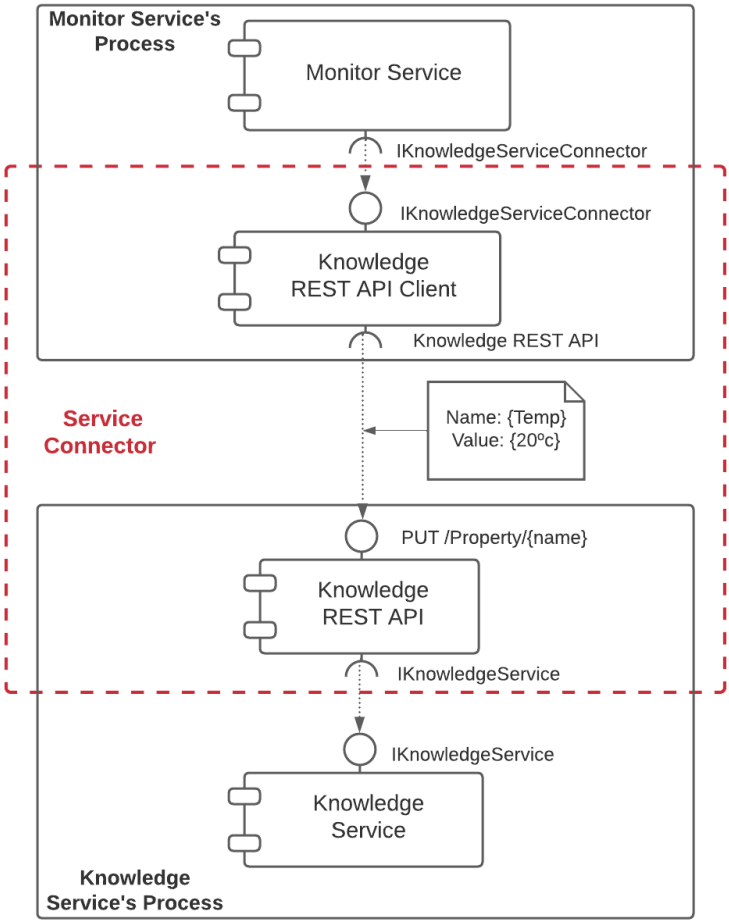


Figura 4.8: Arquitectura del conector de peticiones síncronas.

4.3.5. Notificaciones

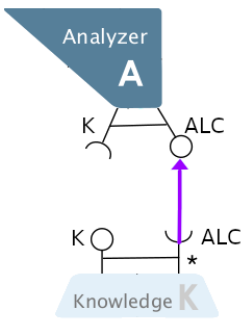


Figura 4.9: Representación inicial del conector entre el servicio de conocimiento y el de análisis.

El siguiente mecanismo de comunicación que tratamos fueron las notificaciones. Recordemos que es de tipo ***broadcast***: consiste en transmitir mensajes desde un servicio a todos aquellos que pertenecen a la capa superior. Cada uno decidirá si debe procesarlo o no. Sería ideal entonces priorizar el **desacoplamiento** entre el emisor y los destinatarios.

Para estudiarla, se tomó como ejemplo la comunicación entre el servicio de conocimiento y los aquellos en el nivel del bucle. Cada vez que se modifique una propiedad de adaptación o una clave de configuración, emitirá

una notificación a la capa superior. Así, por ejemplo, el servicio de análisis sabrá que debe evaluar las reglas de adaptación (figura 4.9).

A continuación, analizamos qué tipo de conector era el más indicado. Empezamos descartando GraphQL porque es un protocolo basado en lecturas. Como el objetivo es enviar información a otros servicios, no nos servía. Respecto a RPC y REST, tampoco cumplían nuestras necesidades. Estos protocolos requerirían de conocer la dirección de todos los servicios para poder contactarlos. O, en su defecto, requeriríamos de un intermediario que los tenga registrados.

Por tanto, optamos por implementarlo usando un *broker* de mensajería. Concretamente, siguiendo el patrón ***publish-subscribe***. El servicio de conocimiento publicaría el evento a través del *broker*. Este evento tendrá como *topic* asociado el nombre de la propiedad que ha cambiado. Así, los servicios de la capa superior podrían decidir si suscribirse o no al tipo de notificación. El *broker* añadirá el mensaje en las colas de mensajería de cada uno de los suscriptores. Entonces, podrán procesarlo cuando puedan, de forma asíncrona.

En la tabla 4.4 mostramos la especificación del evento. Se puede apreciar que el mensaje enviado incluye la **información mínima indispensable**: el nombre de la propiedad que ha cambiado. Elegimos esta aproximación porque se trata de una comunicación asíncrona. Desde que se emite el evento hasta que se procesa, el valor de la propiedad puede haber cambiado. De esta forma, obligamos a las reglas a solicitar su valor en el momento en que se evalúen. Siempre se ejecutarán con la información actualizada.

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.

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 madurez e implantación que tiene su homólogo. Por ejemplo, no cuenta un catálogo muy extenso 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 4.1 presentamos la especificación del mensaje de la tabla 4.4. Podemos apreciar similitudes con la especificación OpenAPI (por ejemplo, en el fragmento 5.2). Figuran tanto la estructura del mensaje como su documentación. La mayor

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

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.

```yaml
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
```

Fragmento 4.1: Especificación del evento de integración *PropertyChangedIntegrationEvent* en el lenguaje AsyncAPI.

Arquitectura del conector

Para implementar este mecanismo, 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 notificados del nuevo mensaje y lo procesarán en cuanto puedan. En la figura 4.10 mostramos la estructura de este nuevo conector.

4.3.6. Peticiones asíncronas

El mecanismo de comunicación restante son las **peticiones asíncronas**. Se trata de peticiones de trabajo que un microservicio envía a otro o a sí mismo. Es ideal para procesos costosos, que el destinatario podrá procesar de forma asíncrona. Ambos servicios deberán encontrarse en el mismo nivel de la jerarquía. En nuestra arquitectura tenemos dos casos 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 (figura 4.11).

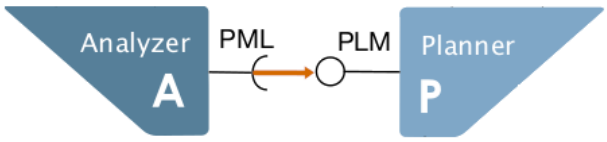


Figura 4.11: Representación inicial del conector entre el servicio de análisis y el planificador.

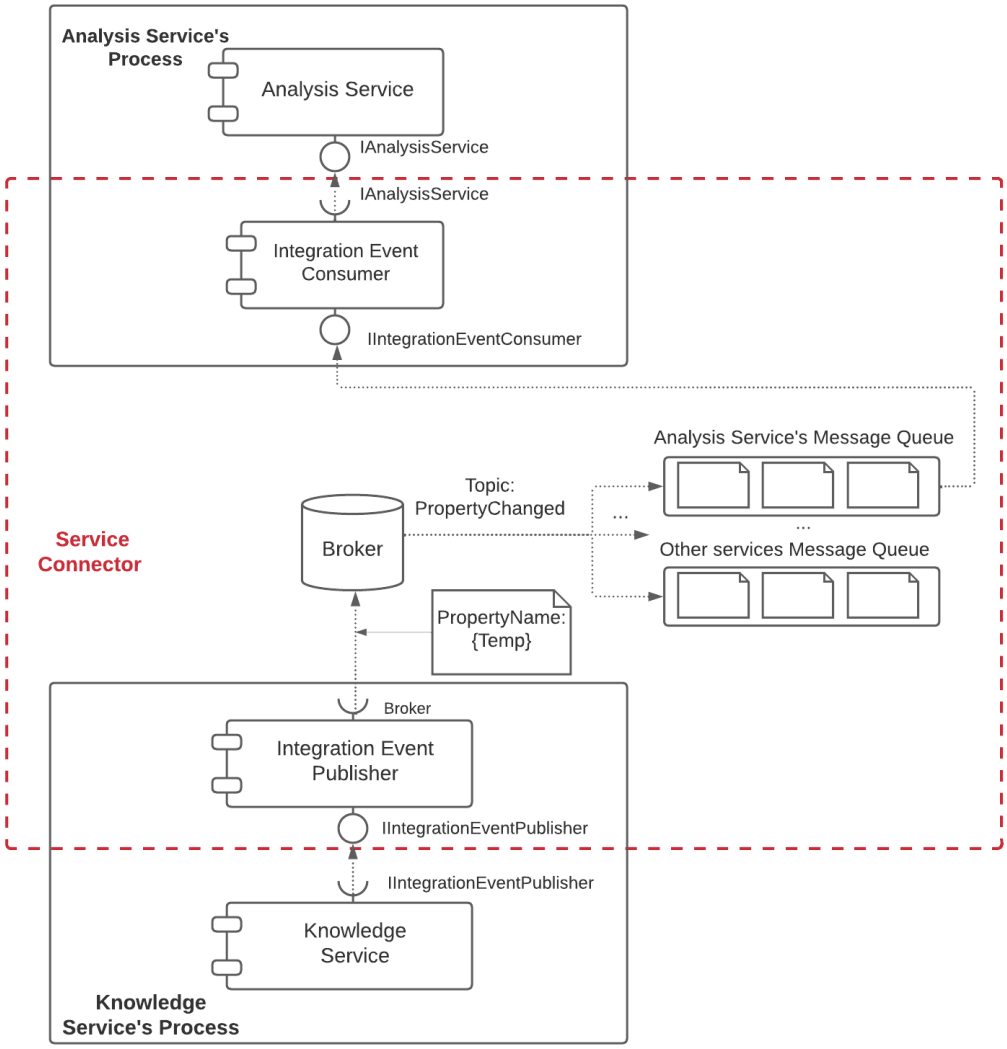


Figura 4.10: Arquitectura del conector para notificaciones mediante un *broker* de mensajería.

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 se la enviará al planificador mediante una petición asíncrona. Podríamos haberla enviado directamente al servicio de planificación, pero preferimos que no se acoplen a un servicio adicional.

A la hora de escoger el tipo de conector, el razonamiento fue muy similar al empleado en las notificaciones. Optamos por implementarlas usando un *broker* de mensajería. En este caso, empleando el patrón de **colas de trabajo**. Los ***workers*** o trabajadores cuentan con una **cola de mensajería específica para las peticiones de trabajo**. El publicador la conoce y, a través del *broker*, envía los mensajes directamente a ella. Los trabajadores los irán recuperando y procesando en cuanto puedan.

En la tabla 4.5 presentamos la especificación de la petición asíncrona para solicitar un cambio de configuración en el 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 planificador recibirá todos los

parámetros que necesita para ejecutar la petición. Aun así, en el momento de ejecución deberá verificar con el conocimiento que algunos de estos no hayan cambiado.

<table>
<tr><td>Nombre</td><td colspan="2">SystemConfigurationChangeRequest</td><td>Cola</td><td>AdaptionLoop.Planification.Requests</td></tr>
<tr><td>Descripción</td><td colspan="4">Petición que representa una propuesta de cambio de la configuración del sistema.</td></tr>
<tr><td rowspan="3">Propiedades</td><td>Timestamp</td><td colspan="3">Fecha y hora de la petición de cambio.</td></tr>
<tr><td>Symptoms</td><td colspan="3">Colección de síntomas que la han desencadenado.</td></tr>
<tr><td>Configuration Requests</td><td colspan="3">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.</td></tr>
<tr><td>Ejemplo</td><td colspan="4">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:
<pre>{
  "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": []
    }
  ]
}</pre></td></tr>
</table>

Tabla 4.5: Especificación de una petición de cambio de configuración del sistema.

AsyncAPI

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[10]. Como mencionamos anteriormente, el estándar todavía se encuentra en desarrollo. Su inclusión está propuesta para la versión `3.0.0` de la especificación.

Arquitectura del conector

La arquitectura de este mecanismo es muy similar a la de las notificaciones. Está compuesta también por tres elementos: un **publicador**, el ***broker*** y un **consumidor**. Pongamos por ejemplo que el servicio de análisis recibe una petición síncrona para cambiar el estado del sistema. Este servicio deberá traducirla a una petición asíncrona y enviarla a través del publicador. A su vez, este enviará el mensaje al *broker* dirigido a una cola determinada.

El *broker* simplemente lo añadirá a la cola especificada. Las instancias del planificador serán notificadas del nuevo mensaje y una de ellas lo procesará en cuanto puedan. En la figura 4.12 mostramos la estructura de este último conector.

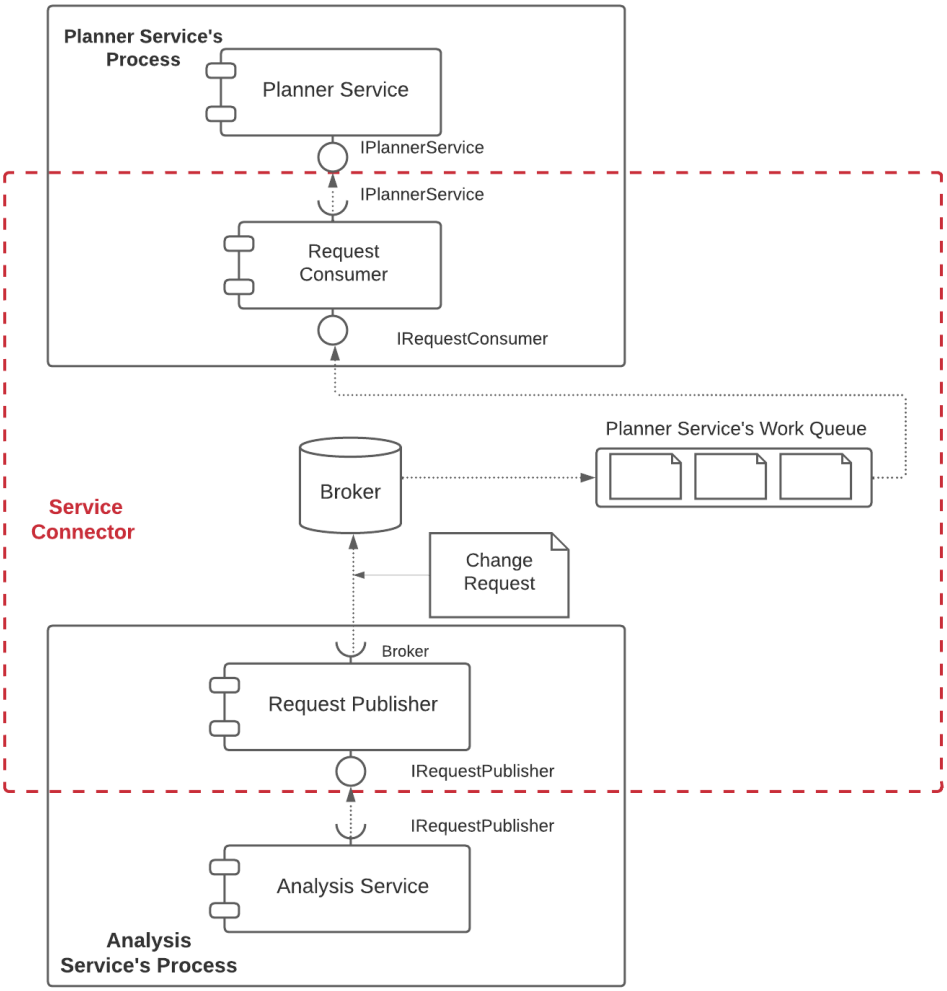


Figura 4.12: Arquitectura del conector para peticiones asíncronas mediante un *broker* de mensajería.

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

4.4 Propuesta arquitectónica

Una vez definidos los componentes y sus conectores, podemos completar nuestra propuesta arquitectónica (figura 4.13). En esta propuesta inicial, cada componente individual representa un tipo de microservicio distinto. Un caso aparte son las sondas y efectores, que pueden desplegarse como parte del recuso manejado.

Por otro lado, los conectores entre ellos están representados mediante distintos tipos de flecha. Podemos comprobar que todas las comunicaciones respetan la jerarquía descrita en la figura 4.3. Los servicios de un nivel superior contactan con aquellos en la capa inmediatamente inferior mediante peticiones síncronas. Los de una capa inferior, con los de la inmediatamente superior a través de notificaciones. Y, finalmente, los de una misma capa mediante peticiones asíncronas. Las APIs que usarán para comunicarse las describimos en detalle en el anexo A - APIs del Sistema.

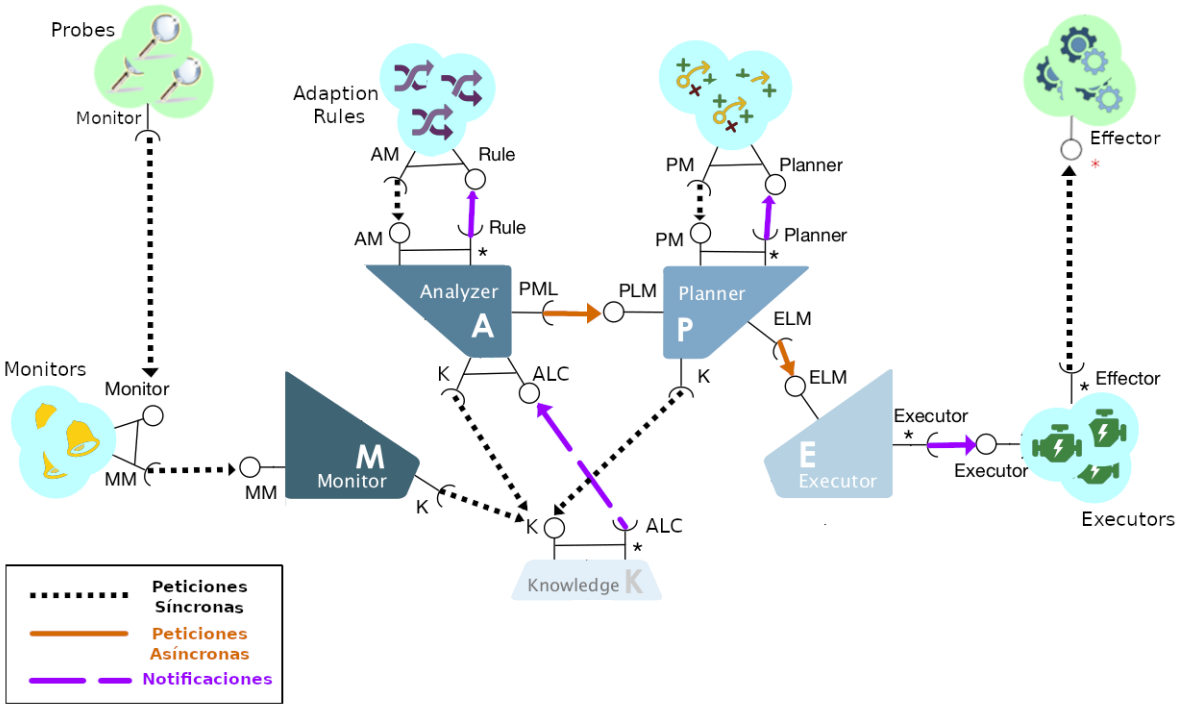


Figura 4.13: Propuesta arquitectónica inicial del bucle MAPE-K distribuido.

CAPÍTULO 5
Implementación

Uno de los objetivos de este trabajo era verificar que la nueva arquitectura fuera viable. La mejor forma de hacerlo es aplicarla en la práctica. En un primer momento consideramos refactorizar el bucle MAPE-K *Lite* para realizar estas pruebas. Pero, debido a su complejidad y las restricciones de tiempo, optamos por implementar una versión reducida del mismo.

Este prototipo se desarrolló a partir de la especificación del sistema existente. Esto nos permitió definir las nuevas APIs de comunicación e implementar la funcionalidad básica. Gracias a él, pudimos evolucionar el diseño según detectábamos nuevas necesidades o problemas que no resolvía nuestra arquitectura. Cuando llegue el momento de la refactorización del bucle original, podremos emplear las APIs desarrolladas.

En este capítulo nos centraremos en la implementación de los microservicios del nivel del bucle y del conocimiento. Dejaremos para más adelante, en el capítulo 6 - Caso de estudio: Sistema de climatización, la descripción de la implementación del recurso manejado. En este capítulo nos centraremos más en la implementación y en las tecnologías empleadas. En el otro, describiremos mediante un caso de estudio cómo encajan los componentes y cómo opera el sistema completo.

La implementación se llevó a cabo en 4 hitos distintos:

5.1 Servicio de monitorización y conocimiento

En esta primera etapa se desarrolló el proceso de monitorización. Este abarca desde que una sonda realiza sus mediciones hasta que se graban en el conocimiento. Esto implicó implementar varios componentes: las sondas y monitores del caso de estudio (capítulo 6), el componente de monitorización del bucle MAPE-K y la base de conocimiento (figura 5.1).

Para su desarrollo se optó por el lenguaje C# y el *framework* ASP.NET[1]. Este *framework* es específico para implementar servidores web. Forma parte de la plataforma .NET de Microsoft. Lo elegimos por nuestra experiencia de desarrollo en ella. 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

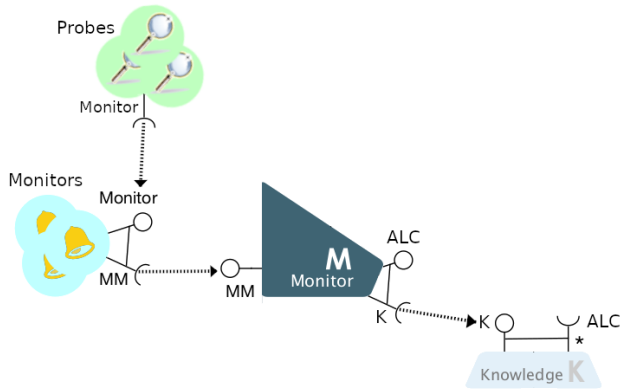


Figura 5.1: Componentes desarrollados durante el primer hito: Sondas, monitores, el módulo de monitorización y el conocimiento.

5.1.1. Peticiones síncronas

En este hito se prototiparon las peticiones síncronas (flechas negras de la figura 5.1). Los servicios que las implementan exponen APIs REST mediante *endpoints* HTTP. Por ejemplo, el servicio de conocimiento expone operaciones que permiten recuperar o modificar propiedades de adaptación. Nos centraremos en la primera, ya descrita en la tabla 4.2.

En el fragmento 5.1 mostramos su implementación. Podemos observar que se trata de un método llamado *GetProperty*. Su lógica es sencilla: busca en un diccionario la propiedad cuyo nombre recibe por parámetro. Si la encuentra, devuelve su valor con un código HTTP (`200 OK`). En caso contrario, sólo devuelve un código de error que describe el motivo de fallo: formato de la petición incorrecto (`400 - Bad Request`) o que no se ha encontrado la propiedad (`404 - Not Found`).

Se puede comprobar que el método cuenta con una serie de comentarios (líneas `1-8`) y atributos (`10-12`). En conjunto estos conforman su documentación. Describen qué hace el método, sus parámetros de entrada y posibles respuestas. OpenAPI puede aprovecharlos para generar una especificación más completa. Resulta entonces 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 propertyFound =
```

```
            properties.TryGetValue(propertyName, out PropertyDTO property);

        if (!propertyFound)
        {
            return NotFound();
        }

        return Ok(property);
    }
```

Fragmento 5.1: Implementación del método `GetProperty` decorado para generar la especificación OpenAPI.[2]

Para generar la especificación empleamos la librería *Swashbuckle.AspNetCore*[3]. Esta es capaz de extraerla de un servicio ASP.NET existente. En el fragmento 5.2, se muestra cómo se describe el *endpoint* en este estándar. Podemos confirmar que se han incluido los comentarios y atributos que documentaban en el código. Por ejemplo, figuran los parámetros, las respuestas y sus códigos HTTP, etc. Es destacable la similitud que presenta con la especificación de la tabla 4.2.

```
"paths": {
  "/Property/{propertyName}": {
    "get": {
      "tags": [
        "Property"
      ],
      "summary": "Gets a property given its name.",
      "parameters": [{
          "name": "propertyName",
          "in": "path",
          "description": "The name of the property to find.",
          "required": true,
          "schema": {
            "type": "string"
          }
        }
      ],
      "responses": {
        "200": {
          "description": "The property was found. Returns the value of the
              property.",
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/PropertyDTO"
              }
            }
          }
        },
        "404": { "description": "The property was not found.", },
        "400": {
          "description": "There was an error with the provided arguments.",
        }
      }
    }
  }
```

Fragmento 5.2: Especificación OpenAPI del método para obtener una propiedad del conocimiento (GetProperty). [4]

[2]Código disponible aquí.
[3]Página oficial: https://github.com/domaindrivendev/Swashbuckle.AspNetCore
[4]Código disponible aquí.

A partir de la especificación, esta librería añade al servicio una interfaz de usuario. Esta es accesible a través del *endpoint* `/swagger`. Allí, se nos servirá una página web con el listado de todas las operaciones que ofrece el servicio (figura 5.2). De cada una nos muestra su documentación, sus parámetros, etc. Incluso permite ejecutarlas. Así, puede ser de ayuda a los desarrolladores que necesiten trabajar con esta API.

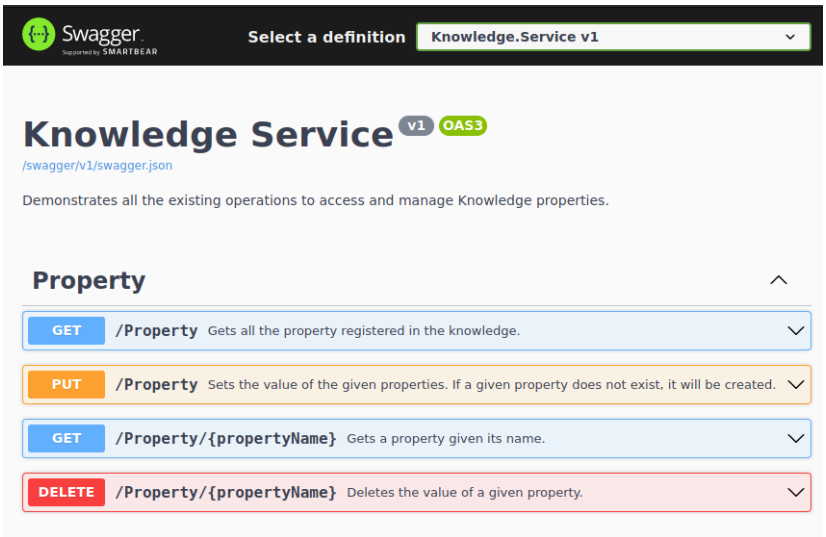


Figura 5.2: Interfaz de usuario ofrecida por Swagger para el servicio de conocimiento. Se genera a partir de la especificación OpenAPI.

Por otro lado, también se pudo generar el API *client*. Como comentamos en la sección 4.3.4 - Open API, tenemos gran variedad de generadores de código a nuestra disposición. Nosotros optamos por la librería `OpenAPI.Generator`[5]. En concreto, por un generador de código de `C#`. Usándolo, pudimos obtener una librería que permite contactar con el servicio, sin necesidad de implementar mucho código. Por ejemplo, el componente de monitorización del bucle contacta con el conocimiento a través de un API *client*.

Además, podremos utilizar estas herramientas durante la refactorización del bucle MAPE-K original. Aunque este está desarrollado en Java, podremos generar el código a partir de la especificación de nuestra implementación. Tanto de los clientes como los servidores. Esto ayudará a agilizar enormemente el proceso de desarrollo.

5.1.2. Componentes: Módulos de monitorización y conocimiento

Los componentes implementados en este hito son muy sencillos. Simplemente validan las mediciones de las sondas y extraen de ellas las propiedades de adaptación. Ya hemos hablado de la implementación del servicio de conocimiento. Este ofrece simplemente operaciones de lectura y escritura sobre las propiedades de adaptación y las claves de configuración de los recursos manejados.

Por encima de este, tenemos al servicio de monitorización. En nuestra implementación, actúa como intermediario entre los monitores de la solución y el conocimiento. Ofrece *endpoints* útiles para los monitores, como puede ser reportar sus mediciones o la lectura del conocimiento. De esta forma, los monitores de solución podrán informarse para determinar si una medición es válida o no. Para conectar todos estos servicios se emplearon las peticiones síncronas, que describiremos anteriormente.

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

5.2 Servicio de análisis y reglas

La siguiente etapa que desarrollamos fue la evaluación de las reglas de adaptación. Esta requería implementar el servicio de análisis del bucle MAPE-K y los servicios de reglas de la solución. También necesitamos el mecanismo de las comunicaciones ascendentes: las notificaciones. Con ellas evitamos que los componentes se acoplaran a los de la capa superior.

5.2.1. Notificaciones

Comenzaremos describiendo el desarrollo de las notificaciones (flechas moradas de la figura 5.3). Como ya se describió en la sección 4.3.5, este mecanismo se implementó mediante un *broker* de mensajería. Se eligió `RabbitMQ`[6] para el proyecto, un *broker* sencillo y ampliamente utilizado. [22]

Para implementar los publicadores y consumidores de nuestro conector, utilizamos una librería llamada `Rebus`[7]. Esta nos permitía interactuar con el bus abstrayéndonos de su tecnología concreta. Así, podríamos cambiar de tecnología de transporte en cualquier momento sin tener que modificar nuestros servicios.

Finalmente, para desacoplar la funcionalidad del servicio de la publicación y consumición de mensajes del bus, empleamos `MediatR`[8]. Esta librería implementa el patrón mediador[9]. Permite propagar mensajes dentro de un mismo proceso, sin necesidad de que el emisor ni el receptor se referencien. Para ello, se definen uno o más manejadores (*handlers*) que capturan y procesan el mensaje. En el caso de las notificaciones propagaremos eventos de integración.

Para describir la implementación nos centraremos en la comunicación entre el módulo de conocimiento y el servicio de análisis. Una vez se confirma la escritura de una propiedad de adaptación en el conocimiento, este debe notificar a los servicios en la capa superior. Para ello, comienza propagando internamente un **evento de integración** usando el mediador (línea 11 del fragmento 5.3).

```
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));
13 }
```

Fragmento 5.3: Implementación del método que asigna valor a una propiedad. Muestra un ejemplo de propagación interna de eventos de integración.[10]

El mediador determina que el componente publicador es el destinatario de este evento y lo invoca. Para detectarlo, se basa en las interfaces que implementa (línea 2 del fragmento 5.4). Este componente publica el evento en el bus de mensajería (línea 15). Rebus,

[6] Página oficial: https://www.rabbitmq.com/
[7] Página oficial: https://github.com/rebus-org/Rebus
[8] Página oficial: https://github.com/jbogard/MediatR
[9] Patrón mediador: https://refactoring.guru/design-patterns/mediator
[10] Código disponible aquí.

en base a la configuración del servicio, lo enviará a nuestra instancia de `RabbitMQ`. Todos los suscriptores de este evento recibirán el mensaje 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 }
```

Fragmento 5.4: El publicador de eventos captura el evento de integración y lo publica en el bus.[11]

Finalmente, en el servicio de análisis, se encuentra el componente consumidor (fragmento 5.5). `Rebus` obtiene el evento de la cola de mensajería y se lo transmite. Este lo recibe y lo propaga internamente en el servicio (línea 20). Todos los *handlers* del evento lo recibirán y podrán tratarlo. En este caso, las reglas de adaptación.

```
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 }
```

Fragmento 5.5: El consumidor recibe el evento de integración del bus y lo propaga internamente. Todos los *handlers* de este evento lo recibirán.[12]

5.2.2. Componentes: Servicio de análisis y módulos de reglas

En este hito se desarrollaron los servicios de análisis y de los módulos de reglas (figura 5.3). En cuanto al del módulo de análisis, este realmente no cuenta con mucha lógica. Participa como intermediario entre el conocimiento y los servicios de reglas. Como hemos visto, recibe los eventos de cambios en las propiedades y los propaga a la capa superior. Ofrece operaciones de solo lectura del conocimiento. Así, impedimos las escrituras por parte de las reglas.

[11] Código disponible aquí.
[12] Código disponible aquí.

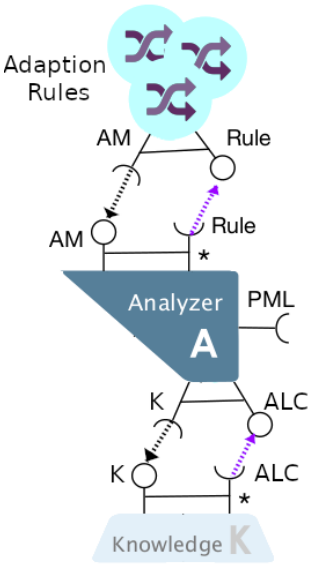


Figura 5.3: Segundo hito: reglas de adaptación y módulo de análisis.

Más tarde nos dimos cuenta que se podría considerar como un servicio "anémico"[13]. [39] No tiene apenas lógica propia. Cuando se realice la refactorización del bucle original, se podría evaluar si tiene sentido mantener esta división. Si no es así, debería desplegarse como una librería que consuman todos los servicios de reglas de adaptación. 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, a continuación ofrecemos una implementación de referencia. En el fragmento 5.6 mostramos la clase base de las reglas de adaptación. Se desarrolló siguiendo el patrón plantilla (o *template*)[14]. 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.

Vemos además que está suscrita a los eventos de integración de cambio de propiedad de adaptación y cambio en la configuración del sistema (líneas 2-3). Cuando el consumidor reciba uno de ellos, lo propagará internamente. Todas las reglas afectadas lo capturarán y se evaluarán.

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

Fragmento 5.6: Clase base para implementar reglas de adaptación. Se evalúa la condición, y si esta se cumple, se ejecuta. [15]

[13] Similar a los objetos anémicos, sin apenas lógica: https://www.martinfowler.com/bliki/AnemicDomainModel.html

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

[15] Código disponible aquí.

Las reglas deben indicar de qué propiedades o claves de configuración dependen. Para ello, hemos implementado una serie de atributos que decoran sus clases. En el fragmento 5.7 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 }
```

Fragmento 5.7: Las reglas declaran sus dependencias sobre propiedades de adaptación usando atributos. Estos se utilizarán para las suscripciones a los temas de los eventos.[16]

En base a los atributos, el servicio se suscribirá a los *topics* de las notificaciones que emite el módulo de análisis. Para leerlos emplearemos la **reflexión**: analizaremos el ensamblado buscando las reglas y obtendremos los valores de sus atributos. Se le pasarán a `Rebus`, que gestionará la suscripción con `RabbitMQ`.

5.3 Servicio de planificación y peticiones de cambio

En el tercer hito se implementaron las peticiones de cambio en la configuración del sistema. Este proceso comienza con la ejecución del cuerpo de las reglas y acaba con la generación de un **plan de cambio**. Esto requirió del desarrollo de un nuevo servicio: el planificador (figura 5.4). En este hito también surgió la necesidad de las peticiones asíncronas, el tercer protocolo de comunicación.

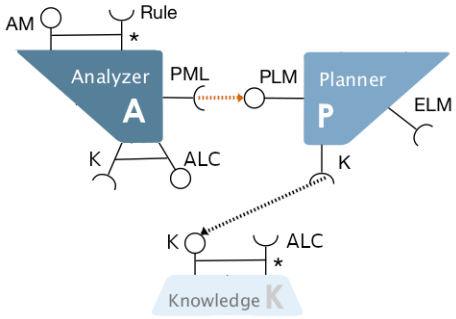


Figura 5.4: Componente desarrollado durante el tercer hito: Planificador.

5.3.1. Peticiones asíncronas

Las peticiones asíncronas son el tercer mecanismo de comunicación de nuestra arquitectura (flecha naranja en la figura 5.4). Estas permiten interactuar a los servicios que pertenecen a una misma capa. Para describir su implementación tomaremos de ejemplo las peticiones de cambio de las reglas de adaptación.

[16] Código disponible aquí.

Cuando la regla evalúa su condición y estima que es necesaria una acción correctiva, se ejecuta su método `Execute`. Este método envía una petición que describe cuál debería ser el siguiente estado del recurso manejado. Por ejemplo, qué componentes deben estar presentes o no, qué conexiones deben existir, etc.

Las reglas comunicarán esta solicitud al módulo de análisis mediante una petición síncrona. Para simplificar el proceso, desarrollamos un *builder*[17] de peticiones por encima de su API *client*. En el fragmento 5.8 mostramos un ejemplo. Allí se indica cuál debería ser la siguiente configuración para el servicio `Climatisation AirConditioner Service` (líneas 7-14). Deberá estar activo (línea 10) y su propiedad `Mode` deberá tener el valor `Cooling` (líneas 11-13). También se incluye 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 }
```

Fragmento 5.8: Implementación de la misma petición siguiendo el patrón *builder*.[18]

El servicio de análisis recibirá la petición y la redirigirá al planificador 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: en lugar de publicarlo en un *exchange*, el mensaje se publicará directamente en la cola de trabajo. Sólo lo debería procesará un servicio.

Como la comunicación es tan similar, únicamente cambiará la implementación del publicador. La mayor diferencia se puede apreciar en el fragmento 5.9. Allí, vemos que el mensaje se enruta directamente a la cola `PlanningServiceQueue` (línea 15). También cambiarán las interfaces que deban implementar los componentes. En este caso `IRequestPublisher` en lugar de `IIntegrationEventPublisher` (línea 2).

```
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)
13   {
14       await _bus.Advanced.Routing.Send(
15           AdaptionLoopPlanningConstants.Queues.PlanningServiceQueue,
```

[17] Patrón *builder*: https://refactoring.guru/design-patterns/builder
[18] Código disponible aquí.

```
16            request);
17
18        return Unit.Value;
19    }
20 }
```

Fragmento 5.9: Las peticiones asíncronas se publican a una cola determinada.[19]

5.3.2. Componentes: Servicio de planificación

El planificador consumirá esta petición de cambio y deberá elaborar un **plan de adaptación**. Para ello, deberá determinar qué acciones son necesarias para alcanzar el estado deseado. Lo comparará con el estado actual, almacenado en el conocimiento, y añadirá al plan las **acciones de adaptación** requeridas. Si el sistema ya estuviera en ese estado, el plan de cambio se quedará vacío y no se propagará.

Por ejemplo, en el fragmento 5.10 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"
    }
  ]
}
```

Fragmento 5.10: Plan de adaptación generado para la regla anterior. Solo contiene una acción de adaptación: cambiar la configuración `Mode` del servicio `AirConditioner`.

Para reducir el alcance del proyecto, no validaremos la viabilidad del plan de adaptación. Se descartó implementar los planificadores específicos de la solución. Sólo contaremos con la estrategia por defecto: validar que el plan contenga al menos una acción.

5.4 Servicio de ejecución y efectores

En el hito final desarrollamos la ejecución del plan de adaptación. Cerramos así el ciclo del bucle de adaptación. Esto requirió del módulo ejecutor, los ejecutores de la solución y los efectores del recurso manejado (figura 5.5).

[19] Código disponible aquí.

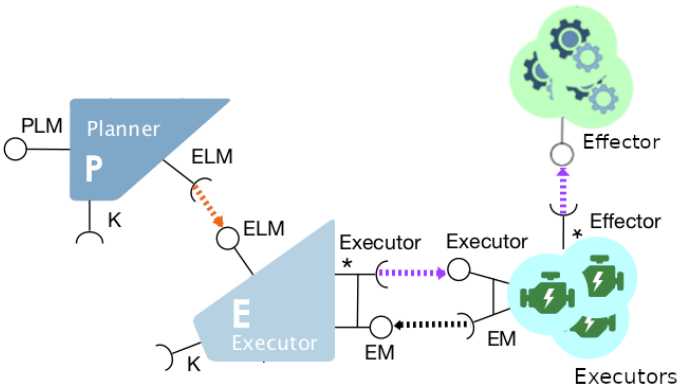


Figura 5.5: Componentes desarrollados durante el cuarto hito: Módulo de ejecución, ejecutores y efectores.

5.4.1. Componentes: Servicio de ejecución, ejecutores y efectores

El servicio de ejecución recibe el plan de adaptación del planificador mediante una petición asíncrona. Para ejecutarlo, deberá agrupar las acciones por el servicio afectado y distribuirlas entre los distintos ejecutores de la solución. Para transmitirlas, enviará cada grupo de acciones mediante una notificación. Pueden existir varios ejecutores en una misma solución y no deberíamos acoplarnos a ellos. El evento enviado es muy similar al que ya mostramos en el fragmento 5.10.

Cuando un ejecutor capture esta notificación, procesará las acciones asignadas. Este componente será el encargado de traducir las acciones de adaptación a manipulaciones de efectores del recurso manejado. Dependiendo del tipo de acción, el efector hará una acción u otra: desplegar un servicio, o eliminarlo, cambiar la configuración, etc. Para reducir el alcance del proyecto, sólo implementamos las adaptaciones de tipo *set parameter*.

En cuanto a la comunicación con el efector, 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 protocolos que ofrezca el recurso (HTTP, mensajería...).

Continuando con el ejemplo del aire acondicionado, el ejecutor recibiría la notificación con las acciones a ejecutar. En este caso, cambiar el modo de funcionamiento a enfriar. Para ello, enviará la acción al efector y se ejecutará. Una vez se confirme el cambio, una sonda lo detectará y notificará a su monitor. Se seguirá el mismo proceso que con las demás mediciones para guardarlo en la base de conocimiento.

CAPÍTULO 6

Caso de estudio: Sistema de climatización

Para validar la arquitectura definida, decidimos implementar un pequeño sistema autoadaptativo. Se trata de un sistema de climatización que gestiona la temperatura de una habitación. En este capítulo especificaremos sus componentes y describiremos su desarrollo. Nos servirá como aplicación de referencia para guiar el diseño de futuras soluciones.

6.1 Análisis

Imagen habitacion termometro aireacondicionado

El primer paso fue 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**. Este ofrece **tres modos de funcionamiento**: un modo para calentar la estancia, otro para enfriarla, y un estado neutral (apagado). Además, cuenta 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 de la estancia deberán activar o desactivar el aparato para alcanzarla. Es importante evitar que el aire acondicionado se encienda y se apague constantemente cuando esta se alcance o sobrepase. Por ello, usaremos unas **temperaturas umbrales**. Por ejemplo, ± 4ºc la temperatura de confort.

6.2 Diseño

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

Para describir su 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 el servicio de aire acondicionado se inicia, 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. Valores: `Off` = 0, `Cooling` = 1, `Heating` = 2
Tipo de dato:	Enumerado

Tabla 6.2: Propiedades de adaptación del sistema de climatización.

6.2.3. Monitores:

Necesitamos los monitores que capturarán los datos de las sondas. Estos deberán procesar las mediciones, agregarlas y filtrarlas. Así detectaremos si hubiera errores de medición, datos obsoletos, etc. De esta forma, evitaremos que se lleve a cabo adaptaciones incorrectas.

Tomemos el monitor de las temperaturas `climatisation.monitor.temperature`. Este valida que la nueva medida de temperatura esté a menos de 5ºc de diferencia; o haya más de un minuto de diferencia entre ellas. Como en el ejemplo trabajamos con un aire acondicionado ficticio, le hemos establecido un margen de error grande. Si no cumple una de estas condiciones, la descartaremos. Así 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.
Actualiza propiedades de adaptación:	`temperature`
Acciones:	**SI** $\lvert$`new-temperature` $-$ `temperature`$\rvert \leq 5{,}0$ **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`.
Actualiza propiedades de adaptación:	`airconditioner.is-deployed, airconditioner-mode, target-temperature, cold-temperature-threshold, hot-temperature-threshold`
Acciones:	**SI** `property` $\neq$ `new-value` **ACTUALIZA-KNOWLEDGE** `property` $=$ `new-value`

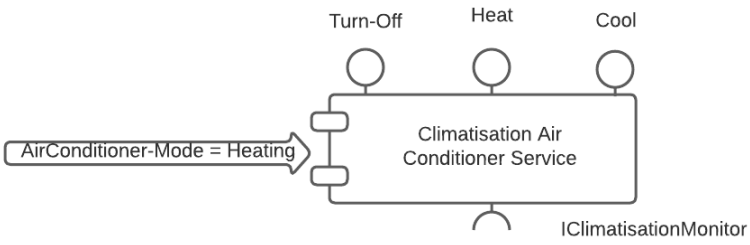
Tabla 6.3: Monitores del bucle MAPE-K del sistema de climatización.

6.2.4. Reglas de adaptación

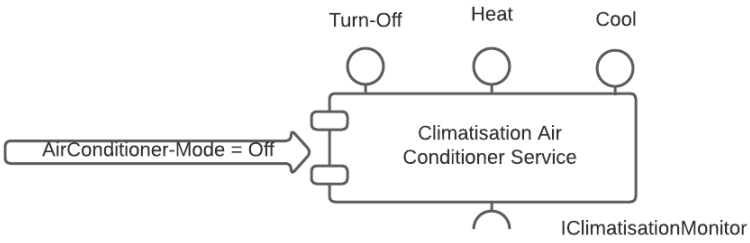
Las reglas de adaptación de nuestro sistema deberán activar o desactivar el aire acondicionado en base a la temperatura actual. Por ejemplo, si la temperatura es inferior al umbral de frío, el aparato se enciende en modo calefacción. Las reglas se ubicarán en el servicio `Climatisation.Rules.Service`. Se dispararán cuando cambie uno de sus parámetros.

En la tabla 6.4 especificamos las cuatro reglas necesarias. Como comentamos en el capítulo anterior, en el prototipo implementado del bucle MAPE-K nos limitamos a implementar las adaptaciones de tipo *set parameter*. Por tanto, no definiremos ninguna regla con adaptaciones de despliegue o de *binding*.

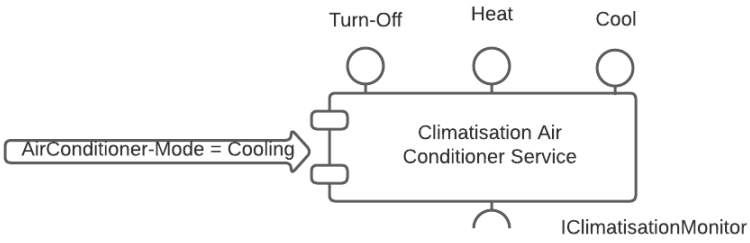
Regla: *EnableAirConditionerHeatingModeWhenColdTemperatureThresholdExceeded*
Descripción: Activa el aire acondicionado en modo calefacción cuando la temperatura sea inferior al umbral de frío.
Condición: *airconditioner-mode* $\neq$ *Heating* **AND** *temperature* $\leq$ *cold-temperature-threshold*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



Regla: *DisableAirConditionerWhenHeatingModeEnabledAndTargetTemperatureReached*
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* $\geq$ *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* $\neq$ *Cooling* **AND** *temperature* $\geq$ *hot-temperature-threshold*
Cuerpo:



Regla:	*DisableAirConditionerWhenCoolingAndTargetTemperatureReachedAdaptionRule*
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* $\leq$ *target-temperature*
Cuerpo:	

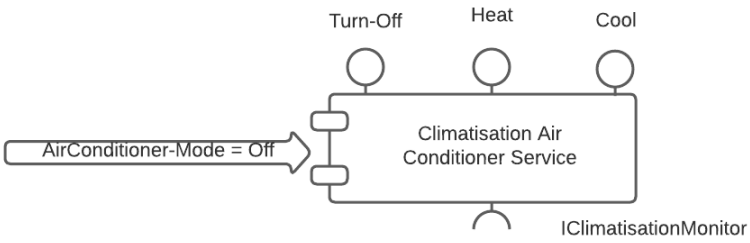


Tabla 6.4: Reglas de adaptación del sistema de climatización.

6.2.5. Efectores:

Los efectores que expondrá el aire acondicionado para cambiar su modo de funcionamiento son:

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.

6.3 Implementación

En esta sección describiremos la implementación del sistema de climatización. Para funcionar como un elemento autónomo, tendremos que emplear los servicios del bucle MAPE-K distribuido y otros cuatro nuevos. Por un lado estará el recurso manejado, el aire acondicionado. Y por el otro, tendremos los servicios de infraestructura necesarios para operar con el bucle. Estos son el monitor, el servicio de reglas y el servicio de efectores.

Para su implementación, hemos empleado las mismas tecnologías descritas en el capítulo 5: microservicios `ASP.NET`, comunicación mediante APIs REST basadas en HTTP y *brokers* de mensajería `RabbitMQ`. A continuación, describiremos cada uno:

6.3.1. Servicio de aire acondicionado

El servicio de aire acondicionado será nuestro recurso manejado. Este cuenta con tres modos de operación: apagado (`OFF`), enfriando (`COOLING`) o calentando (`HEATING`). Según el modo, simulará el aumento o disminución de la temperatura que reporta el termómetro interno. Incluso cuando esté apagado, la temperatura variará para forzar su activación. De esta forma, podemos verificar más rápido si se aplican correctamente las adaptaciones.

El componente expone una API con tres *endpoints* para cambiar el modo. Son estos los que invocará el ejecutor durante las adaptaciones. Además, cuando se produzca el cambio, reportará automáticamente al monitor su nuevo valor. Esto servirá para confirmar el cambio y completar el proceso de adaptación.

El termómetro interno está implementado como un servicio en segundo plano dentro del aire acondicionado. Un *background service*[1] de ASP.NET. Este reportará periódicamente, cada pocos segundos, la temperatura actual al monitor.

Durante el arranque del servicio tenemos otra tarea en segundo plano. Esta se ejecuta una sola vez y se encarga de registrar la configuración inicial del recurso manejado. Este dará valor a las propiedades como los umbrales de temperatura o las variables como `is-deployed` en el conocimiento.

6.3.2. Monitor

El monitor de la solución expone un *endpoint* para recabar las mediciones de temperatura. Se encuentra en la ruta `POST Measurement/Temperature`. Cuando reciba un valor deberá validarlo. El monitor descartará aquellos con diferencias mayores a 5ºC y tomados con menos de un minuto de diferencia. En cambio, si es válido, lo envía al servicio de monitorización. Este lo recibirá y lo almacenará en el conocimiento como una 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 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 definidas en el apartado 6.2.4. Las reglas 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. También tendrán que declarar las propiedades de adaptación de las que dependen para suscribirse a sus cambios.

Para describirlas tomaremos como ejemplo la regla `Disable Air Conditioner When Cooling And Target Temperature Reached Adaption Rule`. Esta desactiva el aparato cuando está activo el modo enfriamiento y se ha alcanzado la temperatura de confort. A lo largo de la memoria ya hemos mostrado algunos fragmentos de su código.

La regla describe las propiedades o configuraciones de las que depende mediante atributos. Son los parámetros que requiere para evaluar su condición. Tomemos por ejemplo el fragmento 5.7. Observamos que declara tres dependencias: `airconditioner-mode`, `temperature` y `hot-temperature-threshold`.

En cuanto a la implementación, ya habíamos discutido anteriormente su método `Execute` en el fragmento 5.8. Solo nos queda exponer la implementación de referencia del método `Evaluate`. En el fragmento 6.1 podemos ver la condición. 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.

[1] Documentación oficial: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services#backgroundservice-base-class

```
protected override async Task<bool> Evaluate()
{
  var currentTemperature =
    await _propertyService.GetProperty<TemperatureMeasurementDTO>(
      ClimatisationConstants.Property.Temperature);

  if (currentTemperature is null)
  {
      return false;
  }

  var airConditionerMode = await _configurationService
    .GetConfigurationKey<AirConditioningMode?>(
        ClimatisationAirConditionerConstants.AppName,
        ClimatisationAirConditionerConstants.Configuration.Mode);

  var targetTemperature = await _configurationService
      .GetConfigurationKey<float?>(
        ClimatisationAirConditionerConstants.AppName,
        ClimatisationConstants.Configuration.TargetTemperature);

  return airConditionerMode == AirConditioningMode.Cooling
      && currentTemperature.Value <= targetTemperature;
}
```

Fragmento 6.1: Implementación de referencia del método `Evaluate`. La regla obtiene del conocimiento el estado actual del sistema y determina si debe ejecutarse.[2]

6.3.4. Ejecutores

Finalmente, tenemos el servicio de ejecutores. 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 recurso manejado. El servicio de ejecutores se suscribirá a estas notificaciones de aquellos componentes que gestiona. En este caso, las referentes al aire acondicionado.

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*). Los efectores que exponga el recurso manejado determinará cómo se ejecutarán estas acciones.

Para el servicio de aire acondicionado, cambiar el modo de operación supone invocar a los *endpoints* HTTP que expone. En el fragmento 6.2 mostramos cómo se ejecuta la acción de adaptación para cambiar la configuración `Mode`. En base al nuevo valor, se invocará a un *endpoint* u otro (líneas 14 a 27). Para ello, empleamos el API *client* generado del servicio de aire acondicionado.

```
public async Task<Unit> Handle(
  SetAirConditionerModeRequest notification,
  CancellationToken cancellationToken)
{
    var succeeded = Enum.TryParse(
      notification.Value,
      out AirConditioningMode mode);

    if (!succeeded)
    {
        return Unit.Value;
```

[2] Código disponible aquí.

```
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  }
```

Fragmento 6.2: Implementación de los efectores del aire acondicionado. Invocan a los endpoints HTTP en base a las acciones de adaptación.[3]

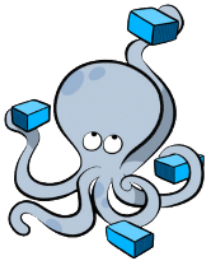
[3] Código disponible aquí.

CAPÍTULO 7
Despliegue y pruebas

En este capítulo describiremos cómo se preparó el despliegue del sistema. Introduciremos también el concepto de observabilidad, que nos permitirá conocer su estado en todo momento. Detallaremos cómo capturamos las métricas de los servicios y maneras de explotarlas. Finalmente, describiremos una serie de pruebas que realizamos para verificar su funcionamiento y el de la arquitectura.

7.1 Despliegue

Debido a la gran cantidad de microservicios que componen la solución, desde muy temprano fue necesario definir un plan de despliegue. Su número iba en aumento y era muy complicado gestionarlos a mano. Para ello, optamos entonces por empaquetarlos en contenedores de `Docker`[1]. Gracias a esto podíamos iniciarlos y pararlos fácilmente. Además que nos aporta una serie de ventajas interesantes: ejecución aislada de los procesos, gestión más fácil de las dependencias, entre otras. [22, 40]



Para orquestar el despliegue de la solución optamos por `Docker Compose`[2]. Nos permite declarar la configuración del despliegue de nuestros servicios. Esto incluye los parámetros de ejecución, número de instancias, políticas de reinicio, etc. Aunque el bucle MAPE-K *Lite* original corre sobre Kubernetes, no necesitábamos un orquestador tan "pesado". Nuestro plan era ejecutar la solución en un único *host*.

`Docker Compose` también nos permite declarar las dependencias entre servicios. Esto fue clave para el despliegue del contenedor de `RabbitMQ`. Debido a que el protocolo requiere de una conexión permanente al bus[41], todos los servicios que dependen de él deben desplegarse después. Por ello, declaramos una dependencia con este servicio y definimos una política de reintentos.

7.1.1. Observabilidad y 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 su estado global en un momento determinado. Especialmente en este caso, en el que participan más de diez microservicios distintos. Si necesitáramos depurar y diagnosticar el comportamiento del sistema, es muy difícil trazar el impacto de una petición.

[1] Página oficial: https://www.docker.com/
[2] Página oficial: https://docs.docker.com/compose/

Por defecto, solo contábamos con los *logs* (registros), que mostramos en la figura 7.1. Aparecen intercalados en una única ventana los de todos los servicios y peticiones concurrentes. Aunque nos pueden resultar utilidad, es una aproximación ineficiente. Incluyen demasiada información y es difícil de procesar para una persona. Además que, según aumente la carga de peticiones, aumentará el número de registros y se volverá más difícil de interpretar.

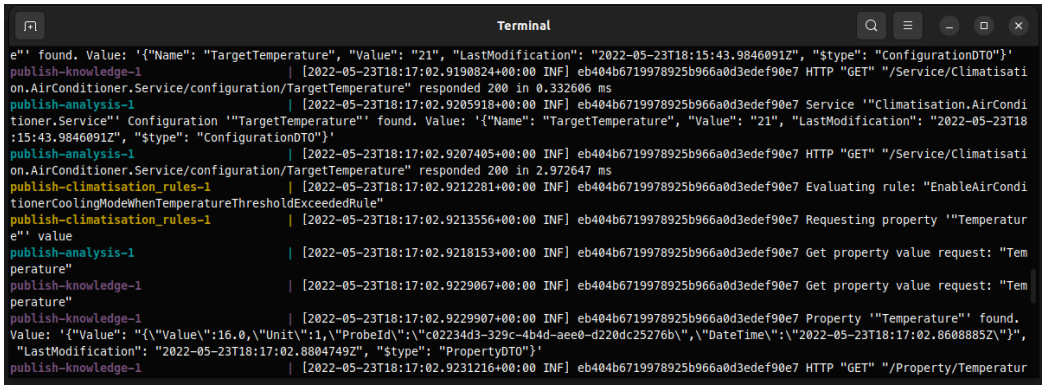


Figura 7.1: Extracto de *logs* de una ejecución habitual.

En el ámbito de los sistemas distribuidos requerimos de soluciones de monitorización y *logging* más avanzadas. [22] Nuestros servicios tendrán que recopilar y reportar datos de su funcionamiento, lo que se conoce como **telemetría**. Esto requerirá de **instrumentar** nuestros sistemas con distintas herramientas o **sondas**. Es exactamente lo mismo que hacemos en la etapa de monitorización del bucle MAPE-K.

Para explotar estos datos recurrimos a técnicas de **observabilidad**. Según [24], la observabilidad «*no es sólo un método para monitorizar sistemas en producción, si no también para ser capaces de entender su comportamiento usando un número relativamente bajo de señales*». Con **señales** se refiere a las distintas fuentes de información de telemetría de las que dispongamos.

La observabilidad nos ayuda a detectar **fluctuaciones en el funcionamiento** de nuestro sistema. Estas puedes ser errores, realentizaciones, caídas de servicios, etc. También nos permite **explicar sus causas** a partir de las señales. De nuestros servicios podemos capturar tres tipos de señales distintos. Todas ellas son complementarias, ya que reflejan el funcionamiento desde distintas perspectivas. Son conocidas como **los tres pilares de la observabilidad**:

- ***Logs***: Se trata de eventos de la aplicación que se registran durante su funcionamiento. Pueden ser simples cadenas de texto o estructuras de datos más complejas. En el segundo caso, se trata de registros enriquecidos con propiedades que les dotan de más contexto. Es el mecanismo de telemetría que ofrece más detalle del funcionamiento de un servicio concreto. También es el más usado.
- **Métricas**: Son datos agregados que nos permiten conocer el estado global de nuestros servicios. [42] Se calculan a partir de mediciones de parámetros del servicio en un momento determinado. Por ejemplo del número de peticiones recibidas, su duración, etc. Normalmente se representan cómo series temporales: peticiones por segundo, duración media de las peticiones, etc.
- **Trazas distribuidas**: Se trata del mecanismo más reciente. Es una forma de registrar el recorrido que hace una petición a través de los distintos microservicios que componen nuestro sistema. Nos permite ver cómo participa cada uno de ellos en la operación y qué impacto tiene en el rendimiento. [24]

> Para registrar una traza, le asignaremos a la petición un identificador único que se propagará con cada sub-petición. Están compuestas por *spans*, operaciones que se realizan dentro de la petición. Cada uno puede tener otras sub-operaciones anidadas. [42]

Para explotar estos datos, necesitaremos entonces poder hacer consultas sobre ellos. Pongamos por ejemplo que hemos detectado que ha aumentado considerablemente la métrica de la duración media de las peticiones. A partir de la fecha y hora de este suceso, deberíamos poder recuperar la información necesaria para responder a la pregunta de qué ha pasado. Ya sean *logs*, trazas u otras métricas relacionadas.

Con este fin se surgen las **plataformas de observabilidad**. Se trata de conjuntos de servicios que capturan los datos de telemetría. Los procesan y almacenan para su posterior consulta. [43] También contamos con servicios que nos permiten visualizar y consultar de forma conjunta toda esta información. Por ejemplo, mediante *dashboards* o paneles.

Plataforma de observabilidad

Para poder capturar y explotar estas señales, necesitaremos construir nuestra propia plataforma de observabilidad. Para este trabajo hemos optado por una combinación de cuatro servicios. `Grafana Loki` para capturar los *logs*. `Prometheus` para capturar las métricas. `Jaeger` para capturar las trazas distribuidas. Y `Grafana` para visualizar y consultar todos los datos.

Para capturar la telemetría, empleamos el estándar **OpenTelemetry**[3]. Se trata de un proyecto desarrollado por la Cloud Native Computing Foundation (CNCF). Tiene el objetivo de definir un mecanismo estándar para recopilar y transmitir datos de telemetría. Para ello, ofrece un conjunto de librerías que nos permite instrumentar nuestras aplicaciones. Podremos enviar estos datos a cualquier plataforma que ofrezca extensiones compatibles.

Con estas herramientas hemos instrumentado todos nuestros servicios. Nuestra plataforma de observabilidad tiene la siguiente estructura (figura 7.2):

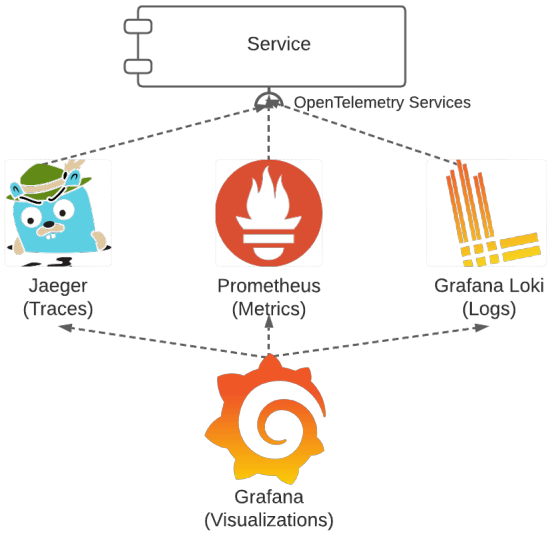


Figura 7.2: Estructura de nuestra plataforma de observabilidad

[3]Página oficial: https://opentelemetry.io/

Grafana Loki: *Logs*

`Loki`[4] es un agregador de *logs* estructurados desarrollado por Grafana Labs. Todos los servicios instrumentados se los enviarán y este los almacenará de forma centralizada. Para facilitar las consultas, Loki indexa todos los registros en base a etiquetas (*labels*), metadatos especificados por el usuario. Por ejemplo, el nivel (información, *warning*...) o el nombre del servicio que los emite.

Nuestros servicios emiten los *logs* siguiendo el mismo convenio. Estos deben incluir la fecha del evento y su nivel de severidad. También incluirán propiedades que indiquen el nombre del emisor y el del entorno en el que se encuentra. Además, queremos correlacionar *logs* de distintos servicios que se originen de una misma petición. Para ello los etiquetaremos con un identificador único: el identificador de la traza (`traceId`). En la figura 7.3 mostramos un ejemplo de toda la información registrada.

Figura 7.3: Ejemplo de la estructura de un registro.

Prometheus: Métricas

`Prometheus`[5] es una herramienta de monitorización y alertas desarrollada originalmente por SoundCloud. Nos permite capturar mediciones de parámetros de nuestros servicios. Estas serán procesadas y almacenadas como series temporales. Sobre ellas, podremos hacer distintos tipos de análisis, consultas y visualizaciones.

Por defecto, ASP.NET captura distintas métricas que podemos exponer con Prometheus. También nos permite definir las nuestras propias. Estas pueden ser de distintos tipos. Los más habituales son los **contadores** e **indicadores**. [24] Los primeros aumentan su valor cada vez que ocurre un evento determinado. Por ejemplo, el número de peticiones recibidas. Por otro lado, los indicadores representan un valor en un momento determinado. Por ejemplo, el número de usuarios activos actualmente.

Para importar los datos, el servidor de Prometheus ejecutará periódicamente consultas HTTP sobre un *endpoint* estándar: `GET /metrics`. Nuestros servicios instrumentados expondrán sus métricas y mediciones allí. En la figura 7.4 tenemos un ejemplo. Muestra las métricas por defecto y algunas definidas por nosotros, como un contador de peticiones de configuraciones.

[4]Página oficial: https://grafana.com/oss/loki/
[5]Página oficial: https://prometheus.io/

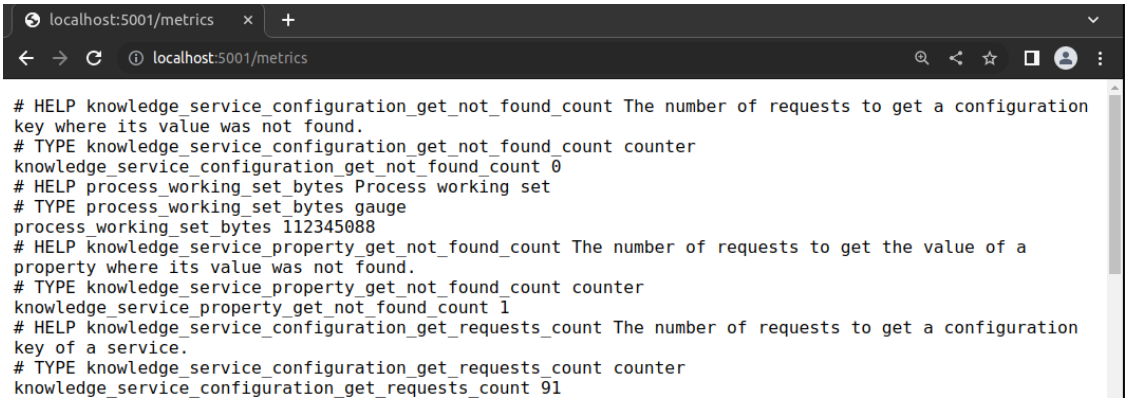


Figura 7.4: Ejemplo de las métricas que expone el *endpoint* de Prometheus en el servicio de conocimiento.

Jaeger: Trazas distribuidas

Jaeger[6] es un sistema para la captura de trazas distribuidas. Fue desarrollado originalmente por Uber Technologies. Todos los servicios instrumentados enviarán allí los fragmentos correspondientes a las actividades en las que participan (los *spans*). A partir de ellas y el identificador común de la traza, es capaz de reconstruir la traza completa de la petición.

Gracias a las trazas distribuidas, podemos ver todas las actividades que desencadenó una petición concreta. Podemos ver sus nombres, su duración e incluso las subactividades en las que derivan. En nuestro caso, mostramos en la figura 7.5 un fragmento de la traza del reporte de una medición de temperatura. Vemos que esta inicia con la actividad de reporte y que acaba desencadenando actividades en seis servicios distintos.

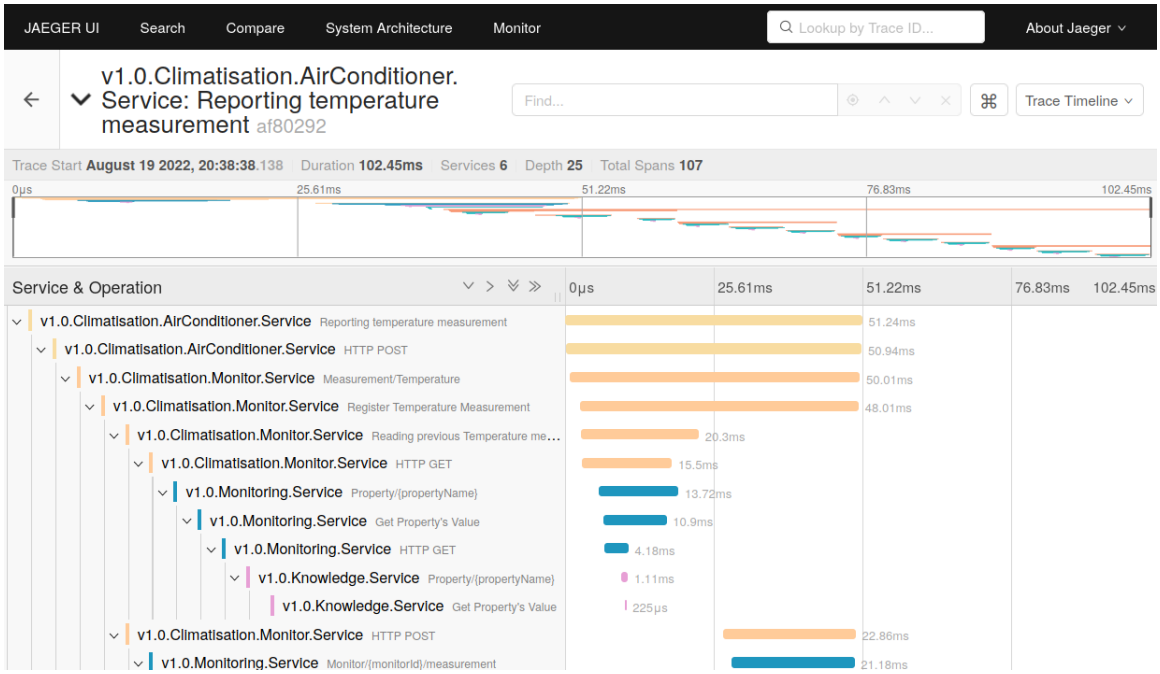


Figura 7.5: Ejemplo de una traza distribuida de Jaeger. Representa las actividades que desencadena el reporte de una medición de tempertura.

[6]Página oficial: https://www.jaegertracing.io

A partir de las trazas, Jaeger también es capaz de inferir la arquitectura de nuestra aplicación. En la figura 7.6 mostramos la arquitectura inferida de nuestro sistema de climatización. Podemos comprobar que la implementación respeta la jerarquía de los microservicios definida en este trabajo. Aunque, este diagrama no muestra el mecanismo de comunicación usado entre cada uno.

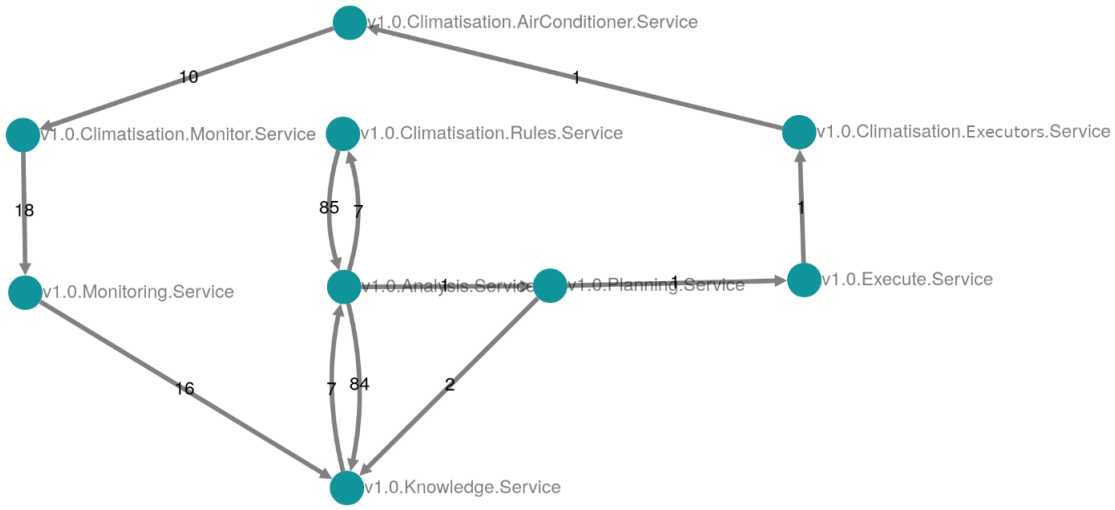


Figura 7.6: Arquitectura inferida por Jaeger de nuestro sistema de climatización a partir de las trazas capaturadas.

Grafana: Visualización

La última pieza del puzzle de observabilidad es Grafana. Desarrollado también por Grafana Labs, es una herramienta para la monitorización y visualización de datos. Gracias a su sistema de *plugins*, es compatible con una gran variedad de fuentes de información: bases de datos, servicios web, servicios de métricas, etc.

Nos permite **explorar los datos** a través de consultas sobre las fuentes de información. Todas ellas se hacen usando los mecanismos ofrecidos por cada plataforma. Este sería el caso de Prometheus, donde podemos usar su lenguaje de consultas `PromQL`. Con él, podremos consultar y visualizar métricas. Esto nos permitirá explotar al máximo las capacidades de cada una.

Incluso, podemos ir más allá y **definir relaciones entre datos de fuentes distintas**. Retrocedamos a la figura 7.3, que representa los *logs* de Loki. Si nos fijamos, en el campo `TracerId` aparece un enlace a Jaeger. Al hacer clic en él, se desplegará en un panel lateral la traza de la petición a la que pertenece el registro. Todo esto nos será de gran ayuda a la hora de **investigar** los motivos de las fluctuaciones en el funcionamiento.

También se pueden aprovechar las consultas de las fuentes de datos para crear **paneles de monitorización**. Esto nos permitirá agregar en solo lugar a los *logs*, las métricas y las trazas. A partir de ellos podemos crear todo tipo de visualizaciones útiles. Por ejemplo, del estado de las peticiones concurrentes, del número de errores, etc. En la figura 7.7 enseñamos nuestro panel de monitorización. Este nos muestra parámetros como la temperatura actual de la habitación (gráfica superior izquierda), un listado de las adaptaciones ejecutadas (panel de *logs* bajo las temperaturas) o información técnica de los servicios (consumo de RAM, tiempo de CPU...). En la siguiente sección las explicaremos en más detalle.

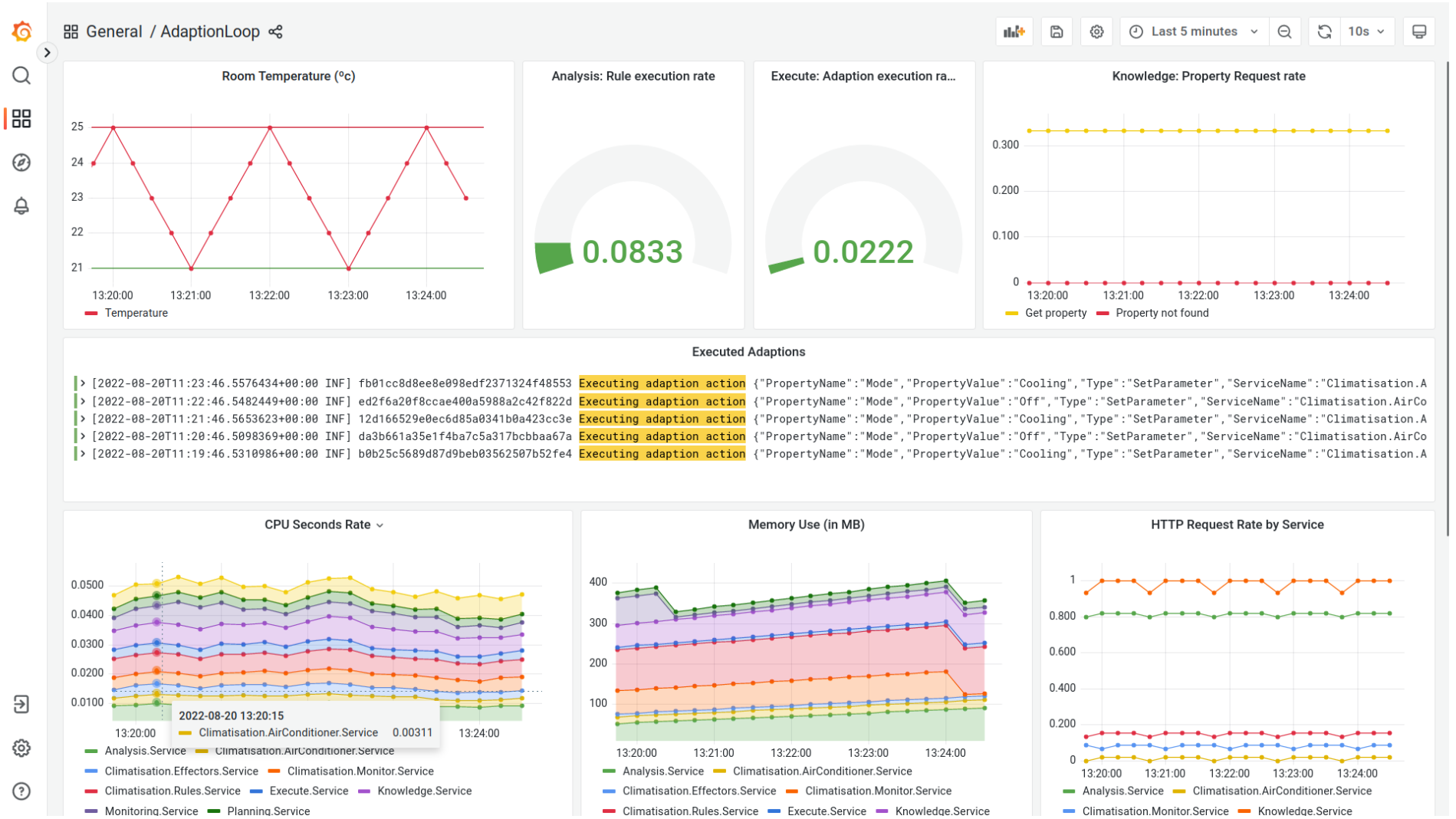


Figura 7.7: Panel de monitorización para la solución autoadaptativa de climatización.

7.2 Pruebas

Finalmente, llegó el momento de poner a prueba nuestro sistema. Queríamos determinar si la arquitectura diseñada era viable o requería de algún refinamiento. Recordemos que el objetivo es aplicarla en el bucle MAPE-K *Lite* original mediante una refactorización. Con esto en mente, diseñamos distintas pruebas para verificar su funcionamiento. Nos resultó de gran ayuda nuestra plataforma de observabilidad, que nos permitirá investigar distintas áreas de la solución.

Las primeras pruebas que ejecutamos fueron las relacionadas con el funcionamiento del sistema de climatización. Hasta que este no se estabilizará, no podíamos emitir ningún juicio sobre la arquitectura. En estos tests verificamos que, a partir de las mediciones de temperatura, debe ser capaz de completar el proceso de adaptación. Esto implica que todas las etapas del bucle MAPE-K se ejecutan correctamente.

Completadas estas pruebas pasamos a verificar, ahora sí, aspectos de la arquitectura. Gracias a la telemetría que capturamos, pudimos responder a distintas preguntas sobre ella. Por ejemplo, si es correcta la división funcional que hemos elegido o si los mecanismos de comunicación son los adecuados. En base a las respuestas, pautamos una serie de correcciones que se podrían aplicar.

7.2.1. Pruebas sobre el sistema de climatización

La primera prueba del sistema de climatización consistió en comprobar su **correcto despliegue**. Para ello, verificamos que después de este, todos los servicios estén operativos. También analizamos los *logs* en busca de registros de error. Detectamos que durante la inicialización aparecen algunos (primera barra en la figura 7.8). Una vez que se completa, el sistema se estabiliza y estos errores desaparecen (resto de barras). Todos provenían de servicios que dependen del *broker* de mensajería (figura 7.9). Como deben establecer una conexión con él durante el arranque, si no ha completado su despliegue todavía, fallarán.

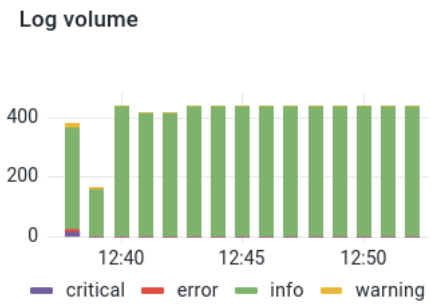


Figura 7.8: Niveles de los *logs* registrados durante la inicialización del sistema.

Según el desarrollador de Rebus, estos errores podrían solucionarse implementado un **servicio en segundo plano que gestione la conexión**.[7] De esta forma, el servicio no fallará durante el arranque e intentará periódicamente conectarse con RabbitMQ. Debido a restricciones de tiempo, en lugar de implementarlo así, optamos por definir una estrategia de reintentos en el fichero de Docker Compose. Así, los afectados se reiniciarán hasta que puedan establecer la conexión correctamente.

A continuación, procedimos a realizar **pruebas sobre la funcionalidad**. En estas, nos centramos en verificar que el sistema **regula correctamente la temperatura** de la habitación. Recordemos que el servicio de aire acondicionado cuenta con un termómetro simulado. Cada 15 segundos, este reporta una medición de temperatura ficticia al monitor de climatización. Esta variará dependiendo del modo activo del aire acondicionado. En base a la medición, el bucle evaluará las reglas y pautará adaptaciones si lo considera necesario.

Comprobaremos entonces que se aplican las cuatro reglas definidas en la tabla 6.4. Todas ellas activan o desactivan un modo del aire acondicionado cuando la temperatura

[7] https://github.com/rebus-org/Rebus.ServiceProvider#delayed-start-of-the-bus

```
[2022-08-20T10:37:28.2214324+00:00 FTL]  Host terminated unexpectedly
Rebus.Injection.ResolutionException: Could not resolve Rebus.Bus.IBus with decorator depth 0 - registrations:
Rebus.Injection.Injectionist+Handler
 ---> RabbitMQ.Client.Exceptions.BrokerUnreachableException: None of the specified endpoints were reachable
 ---> System.AggregateException: One or more errors occurred. (Connection failed)
 ---> RabbitMQ.Client.Exceptions.ConnectFailureException: Connection failed
 ---> System.Net.Sockets.SocketException (111): Connection refused
   at System.Net.Sockets.Socket.AwaitableSocketAsyncEventArgs.ThrowException(SocketError error, CancellationTo
ken cancellationToken)
```

Figura 7.9: Ejemplo de registro relacionado con el fallo al contactar con RabbitMQ durante el arranque.

alcanza un determinado umbral. Por ejemplo, si es muy alta, se debería activar el modo de refrigeración. Cuando se alcance la temperatura de confort, otra regla lo apagará.

Contamos con varias formas de verificarlo. La primera de ellas es mediante la visualización de la temperatura de nuestro panel de monitorización. En la figura 7.10 mostramos la aplicación de las cuatro reglas. Cuando se alcanza uno de los umbrales (las líneas horizontales), el aire acondicionado se activa o desactiva según corresponda.

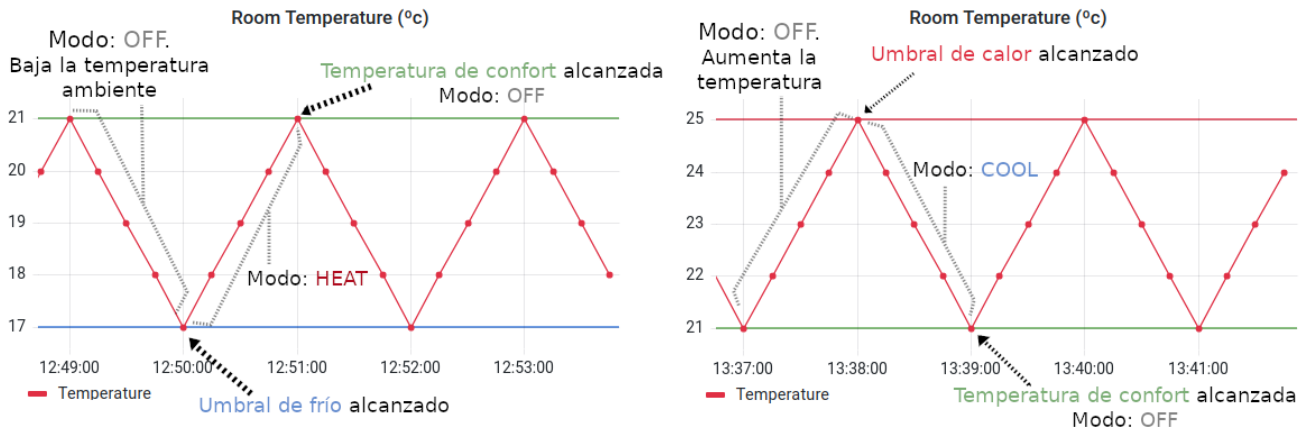


Figura 7.10: Gráficas extraídas de Grafana que muestran el funcionamiento de las adaptaciones. Izq.: Encender y apagar la calefacción. Der.: Encender y apagar la refrigeración.

Otras visualizaciones que nos pueden ser de utilidad son *logs* de los ejecutores de la solución. En el panel de monitorización incluimos aquellos que registran las adaptaciones (figura 7.11). En base a ellos podemos determinar que efectivamente se están pautando las adaptaciones correspondientes. Se intenta cambiar el parámetro `Mode` a `Cooling` o `Off` según corresponda. Para verificar que realmente se está siguiendo el flujo esperado, podemos consultar la traza asociada al registro. En la figura 7.11 mostramos su identificador resaltado.

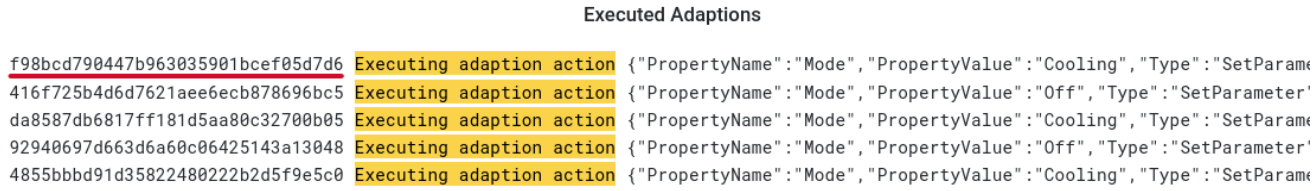


Figura 7.11: *Logs* de los ejecutores de la solución que confirman las adaptaciones pautadas.

7.2.2. Verificación de la arquitectura

Una vez confirmado el correcto funcionamiento del sistema, pudimos proceder a verificar distintos aspectos de la arquitectura. Nos centramos especialmente en la comunicación entre servicios. Gracias a ella, podríamos determinar si era apropiada la división funcional en microservicios que definimos. También nos permitió estudiar el comportamiento de los mecanismos de comunicación elegidos.

Recurrimos en primer lugar a la arquitectura inferida por Jaeger. Recuperamos para ello la figura 7.6. Esta describe el resultado de trazar 10 reportes de mediciones de temperatura. De ellas, 8 pasaron el filtro del monitor, y solo 1 ha provocado la adaptación del sistema. Muestra todos los microservicios que intervinieron y el número de mensajes enviados entre cada uno.

Nuestra primera comprobación fue respecto a la jerarquía de componentes y la dirección de la comunicación. Verificamos que la estructura de este diagrama coincide con la nuestro diseño (figura 4.13). Ningún microservicio aparece conectado con otro no contemplado. También coincide la dirección de la comunicaciones entre ellos.

La siguiente prueba fue sobre el número de mensajes. Si existe un intercambio elevado de mensajes entre dos o más servicios, se puede considerar que son muy "habladores" (*chatty* en inglés). Esto puede ser un indicador de que están muy acoplados. [39] De ser así, pueden convertirse en **puntos de congestión** o **impedir que el dependiente funcione** correctamente si falla el otro. Detectamos dos casos así en el diagrama y los presentamos en la figura 7.12. En verde, aparecen marcadas las conexiones entre los monitores con el servicio de monitorización. Por otro lado, en rojo marcamos las reglas y el servicio de análisis. Como veremos a continuación, son casos muy similares.

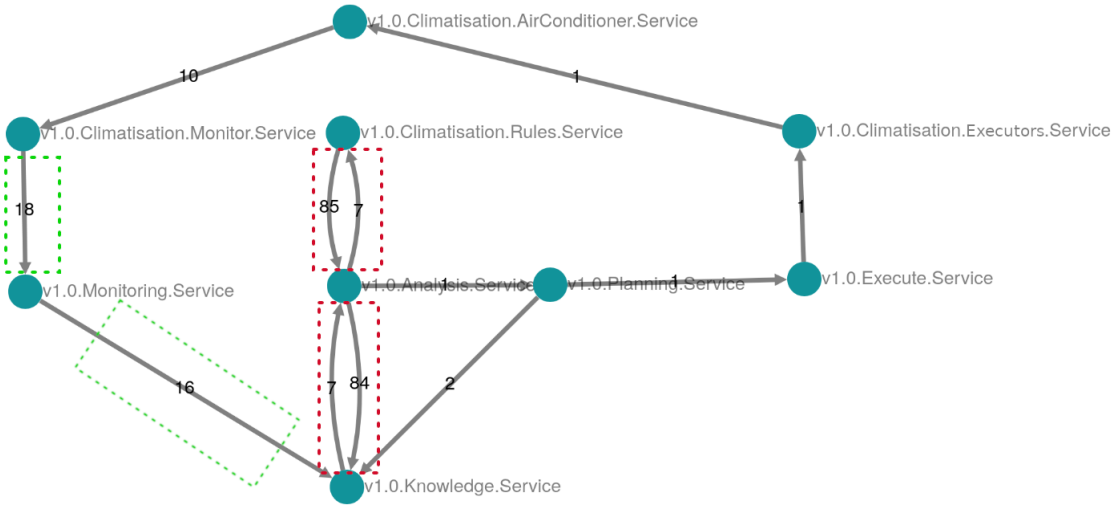


Figura 7.12: Puntos de congestión visibles en la arquitectura inferida por Jaeger. Marcados en verde y en rojo.

El más evidente es el del microservicio de reglas. Realiza 85 peticiones síncronas al módulo de análisis. Este último, a su vez, redirige 84 de ellas al servicio de conocimiento. La faltante podemos asumir que es una petición de cambio de configuración de sistema, resultado de la ejecución de una regla. Presenta entonces muestras muy claras de acoplamiento. En la sección 5.2, ya comentamos que el módulo de análisis actuaría como intermediario entre todas las comunicaciones con las capas inferiores.

El siguiente paso fue comprobar el impacto de esta dependencia mediante **pruebas de carga**. Queríamos ver cómo se comportaba el sistema en casos extremos. Para ello, usamos la librería `NBomber`[8] e implementamos un test sencillo. Durante un minuto, saturará el sistema enviando mediciones falsas de temperatura al monitor. Esto provocará que el bucle esté constantemente ejecutando adaptaciones. Mediremos su impacto mediante el **tiempo medio de adaptación**. Con tiempo de adaptación nos referimos al intervalo que transcurre desde que se reporta la medición hasta que se aplica y confirma una adaptación. Lo calcularemos a partir de la duración de las trazas distribuidas.

En la figura 7.13 presentamos el resultado. A la izquierda aparece nuestro marco de referencia. Fue tomada con la carga habitual del sistema, recibiendo una medición de temperatura cada 15 segundos. Presenta un tiempo medio de 73ms, con picos cercanos a los 150ms. A la derecha se muestra el resultado de una carga extrema: en el espacio de un minuto se enviaron 2312 mediciones. Esto provocó que el tiempo medio aumente hasta los 3.02s, con picos superado los 15s. Tras repetidas ejecuciones confirmamos que los resultados eran muy similares. Confirmamos así nuestras sospechas de la existencia de un punto de congestión.

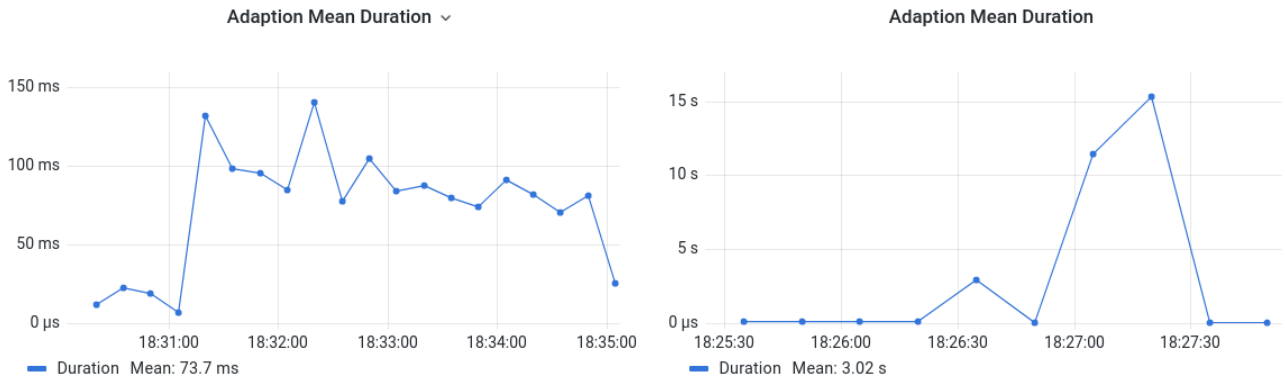


Figura 7.13: Comparación del tiempo medio de adaptación según el nivel de carga del sistema. Izq.: Carga habitual Der.: Carga extrema

Para confirmar dónde se encuentra el punto de congestión, acudimos a las trazas. Analizamos varias peticiones del pico de tiempo medio. En la figura 7.14 mostramos una de ellas con una duración de 16.39s. La mayor parte de este tiempo se encuentra en el intervalo (0.042s - 16.04s), en el que no se ejecuta ninguna actividad. La notificación de cambio de la propiedad temperatura está encolada, a la espera de que el servicio de reglas la procese. Esto confirmó que **se satura y ralentiza el proceso de adaptación**.

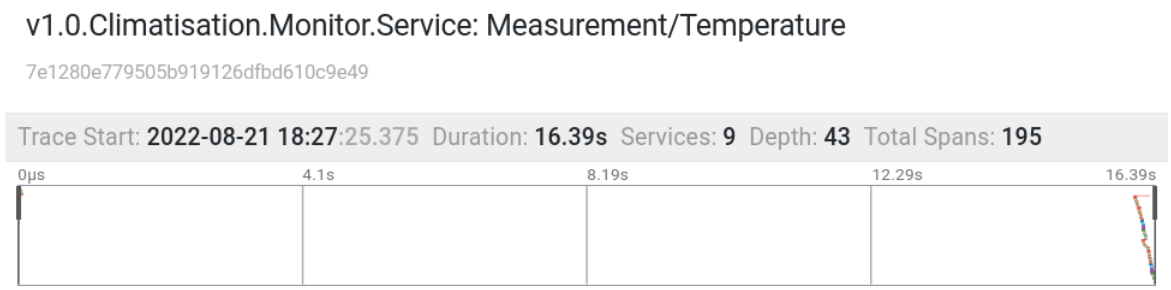


Figura 7.14: Traza distribuida de una adaptación cuando el sistema se encuentra bajo carga extrema.

[8]Página oficial: https://github.com/PragmaticFlow/NBomber.

Respecto al otro sospechoso, la etapa de monitorización, no ha influido en el tiempo medio de adaptación. No hay apenas procesamiento entre que se recibe la medición y esta se descarta o almacena en el conocimiento. Aun así, también presenta un grado de acoplamiento considerable con el servicio de monitorización. En la siguiente sección describiremos algunas propuestas de mejora para nuestro diseño.

7.3 Propuestas de mejora

Una vez realizadas todas estas pruebas, contamos con información suficiente para proponer algunas mejoras a nuestra arquitectura. Estarán orientadas principalmente para solventar el punto de congestión causado por el servicio de reglas. Podrán guiar la refactorización del bucle MAPE-K *Lite* original.

La más sencilla sería **replicar el servicio** de reglas. Podríamos añadir más instancias que se repartan la carga de procesamiento. Esto no sería más que un parche que no soluciona ninguno de los problemas de raíz. Simplemente aplazaríamos un poco el punto de saturación. Además que podría derivar en errores sutiles como adaptaciones incorrectas por el procesamiento en paralelo de estas.

Otra alternativa más adecuada consistiría en **agrupar en un mismo servicio estos componentes tan acoplados**. Por ejemplo, que el módulo de análisis se despliegue como parte del servicio de reglas. Así, la comunicación sería interproceso en lugar de hacer peticiones a través de la red. Esto volvería nuestros servicios más independientes y funcionarán más parecido a los servicios plug & play que describimos en la sección 4.1 - ¿Por qué usar microservicios? En nuestro prototipo este cambio puede aplicarse sin problemas, pero será necesario estudiar si es factible en el bucle real.

Por otro lado, no podemos hacer lo mismo con el componente de conocimiento. Aunque el módulo de análisis presenta mismo grado de acoplamiento, este debe mantenerse como un servicio independiente. Otras etapas del bucle necesitan acceder también a él: los monitores, el planificador, etc. En su lugar, podríamos intentar optimizar la comunicación y **reducir el número de peticiones**. En las reglas de adaptación se solicitaban distintas propiedades de adaptación con peticiones individuales. Estas podrían agruparse en una sola petición que recupere toda la información necesaria.

Actualizamos nuestra propuesta arquitectónica con las dos últimas sugerencias. Esta podría quedar como en la figura 7.15. El módulo de monitorización y el de análisis se desplegarían con los monitores y las reglas respectivamente. Estos podrían publicarse como librerías para que los otros proyectos las consuman.

Respecto al módulo de planificación y los planificadores de la solución, dependerá de cómo sea su implementación en el bucle de adaptación real. A priori, parecen muy similares a los servicios de reglas de adaptación. En ese caso, podría estudiarse también su despliegue conjunto con cada servicio de planificación. Finalmente, el módulo de ejecución iría "por libre". Como los ejecutores no realizan peticiones síncronas, podrían desplegarse por separado. No están acoplados entre si.

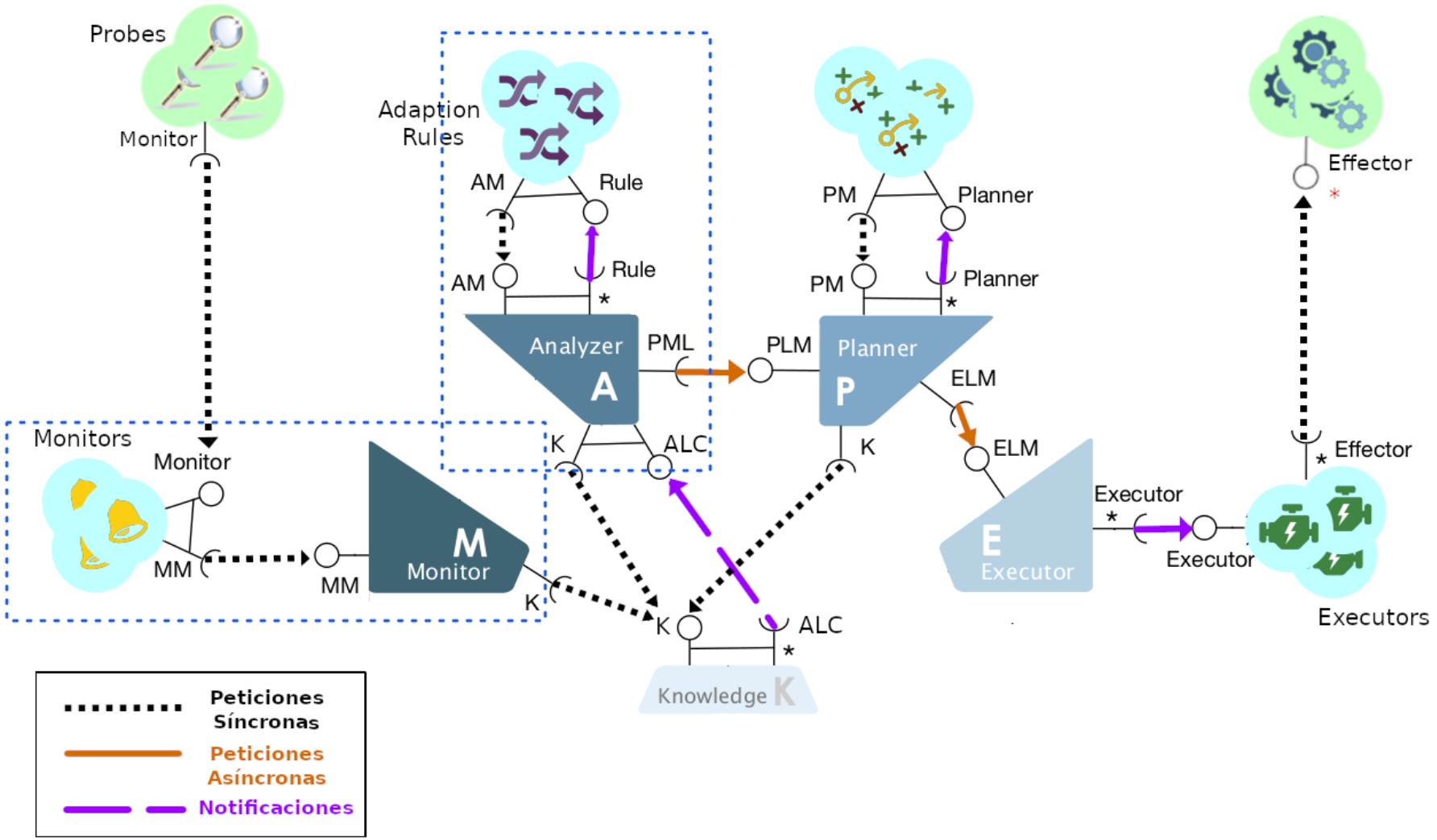


Figura 7.15: Propuesta arquitectónica final del bucle MAPE-K distribuido.

CAPÍTULO 8
Conclusiones

En este capítulo se realiza una retrospectiva del desarrollo del trabajo y presentaremos algunas conclusiones sobre el mismo. Por ejemplo, el grado de realización de los objetivos marcados o la descripción de vertientes todavía abiertas.

Al inicio del trabajo se presentaron una serie de objetivos que se quería alcanzar.

Para el desarrollo del trabajo nos planteamos los siguientes objetivos:

1. Diseñar una arquitectura para soluciones autoadaptativas 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.

8.1 Relación con asignaturas cursadas

El trabajo desarrollado tiene relación con varias asignaturas cursadas durante el máster. Entre ellas, podemos destacar:

- **Diseño de Sistemas Ubicuos y Adaptativos** (SUA): Es la asignatura que más relación guarda con el trabajo. En ella se tratan la computación autónoma y los sistemas autoadaptativos. Mediante el desarrollo de un prototipo de coche autónomo, se presentó el bucle MAPE-K y sus distintas fases.
- **Internet de los Servicios (IoS) y de las Cosas (IoT)** (ISC): En esta asignatura se presentan conceptos relacionados con los servicios web. Se introducen patrones de diseño como las APIs REST y arquitecturas adaptadas a entornos *cloud*. Además, el campo del internet de las cosas se beneficia también de los sistemas autoadaptativos. [17].
- ***Data Science*** (DAS) y **Extracción de información desde la red social** (ERS): En ambas asignaturas se trata la extracción de conocimiento a partir de los datos. Mediante técnicas de obtención, procesamiento y visualización, podemos interpretarlos y "contar una historia" con ellos. Ambas tuvieron una gran influencia en el desarrollo de la plataforma de observabilidad y las visualizaciones implementadas (capitulo 7).

8.2 Trabajos futuros

En cuanto a trabajos futuros, el más evidente es aplicar la refactorización sobre el bucle MAPE-K *Lite* de FaDA. Aunque el prototipo pretendía ser lo más fiel al sistema original, es posible que surjan nuevas dificultades no contempladas. Deberá definirse una estrategia para atacarla e ir implementándola gradualmente. Para ello, podrán aprovecharse las interfaces definidas de los servicios y sus especificaciones en lenguajes estándares como OpenAPI. Mediante la generación de código, tanto de clientes como de servidores, se podrá reducir el tiempo y esfuerzo necesarios para la implementación. abo.

Todavía quedan algunas vertientes abiertas que se podrían explorar. Entre ellas, la implementación de *multitenancy* (multicliente). [21] Es decir, permitir que varias soluciones autoadaptativas empleen la misma infraestructura del bucle; pero de forma segregada, sin poder interferir entre ellas o acceder a los datos de otras. Se excluyó de este trabajo para reducir el alcance del proyecto. Para implementarla, se debería desarrollar mecanismos de autenticación y autorización, que permitan identificar a cada aplicación y limitar sus permisos. Además deberá estudiarse cómo proteger el acceso a la información de los distintos clientes.

Por otro lado, se podrían investigar más maneras de explotar la telemetría recogida por la plataforma de observabilidad. Por ejemplo, para informar al proceso del bucle MAPE-K. De esta forma, se podrían extraer propiedades de adaptación y ampliar nuestro conocimiento del estado del sistema. En base a estas, se podrían definir nuevas reglas que nos permitan adaptar nuestro sistema a distintas situaciones. Un caso interesante sería analizar las métricas de peticiones concurrentes para desplegar nuevas instancias de los servicios.

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. doi: 10.1109/ICSE.2004.1317410.

[2] I. Corporation, "An Architectural Blueprint for Autonomic Computing," tech. rep., IBM, 2006. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.1011&rep=rep1&type=pdf.

[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. doi: 10.1007/978-3-642-02161-9_3.

[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. http://hdl.handle.net/11705/JCIS/2021/023.

[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. http://laputan.org/mud/.

[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. https://standards.ieee.org/standard/42010-2011.html.

[9] D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17, Oct. 1992. doi: 10.1145/141874.141884.

[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. doi: 10.1145/337180.337201.

[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. doi: 10.1109/32.508313.

[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. doi: 10.1007/3-540-45177-3_3.

[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. doi: 10.1145/1186778.1186782.

[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. doi: 10.1145/3241403.3241423.

[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. doi: 10.1016/j.jksuci.2018.01.003.

[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. https://riunet.upv.es/handle/10251/153180.

[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. doi: 10.1016/j.future.2019.09.016.

[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. doi: 10.1007/978-3-319-48746-5_45.

[20] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-Native Applications," *IEEE Cloud Computing*, vol. 4, pp. 16–21, Sept. 2017. doi: 10.1109/MCC.2017.4250939.

[21] H. AlJahdali, A. Albatli, P. Garraghan, P. Townend, L. Lau, and J. Xu, "Multi-tenancy in Cloud Computing," in *2014 IEEE 8th International Symposium on Service Oriented System Engineering*, pp. 344–351, Apr. 2014. doi: 10.1109/SOSE.2014.50.

[22] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Aug. 2021. https://learning.oreilly.com/library/view/building-microservices-2nd/9781492034018/.

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

[24] 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. https://learning.oreilly.com/library/view/distributed-tracing-in/9781492056621/.

[25] "UCI Software Architecture Research - UCI Software Architecture Research: C2 Style Rules." http://isr.uci.edu/architecture/c2StyleRules.html.

[26] 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.

[27] J. Lewis and M. Fowler, "Microservices." https://martinfowler.com/articles/microservices.html, Mar. 14.

[28] 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.

[29] L. Richardson and S. Ruby, *RESTful Web Services*. O'Reilly Media, May 2007. https://learning.oreilly.com/library/view/restful-web-services/9780596529260/.

[30] M. Nally, "REST vs. RPC: What problems are you trying to solve with your APIs?." https://cloud.google.com/blog/products/application-development/rest-vs-rpc-what-problems-are-you-trying-to-solve-with-your-apis/, Oct. 2018.

[31] E. Porcello and A. Banks, *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, Inc., 3rd ed., 2021.

[32] G. Brito and M. T. Valente, "REST vs GraphQL: A Controlled Experiment," in *2020 IEEE International Conference on Software Architecture (ICSA)*, pp. 81–91, Mar. 2020. doi: 10.1109/ICSA47634.2020.00016.

[33] J. Korab, *Understanding Message Brokers*. O'Reilly Media, June 2017. https://learning.oreilly.com/library/view/understanding-message-brokers/9781492049296/.

[34] G. M. Roy, "Chapter 6. Message patterns via exchange routing," in *RabbitMQ in Depth*, Manning Publications, Sept. 2017. https://learning.oreilly.com/library/view/rabbitmq-in-depth/9781617291005/.

[35] IBM, "What are Message Brokers?." https://www.ibm.com/cloud/learn/message-brokers, Jan. 2020.

[36] RabbitMQ, "Publish/Subscribe documentation." https://www.rabbitmq.com/tutorials/tutorial-three-dotnet.html.

[37] OpenAPI_Initiative, "OpenAPI Specification v3.1.0." https://spec.openapis.org/oas/latest.html.

[38] D. Westerveld, "Chapter 3: OpenAPI and API Specifications," in *API Testing and Development with Postman*, Packt Publishing, May 2021. https://learning.oreilly.com/library/view/api-testing-and/9781800569201/.

[39] A. Singjai, U. Zdun, O. Zimmermann, and C. Pautasso, "Patterns on Deriving APIs and their Endpoints from Domain Models," in *26th European Conference on Pattern Languages of Programs*, EuroPLoP'21, (New York, NY, USA), pp. 1–15, Association for Computing Machinery, July 2021. doi: 10.1145/3489449.3489976.

[40] C. De la Torre, B. Wagner, and M. Rousos, *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Corporation, 6.0 release ed., Dec. 2021.

[41] L. Johansson, "Part 1: RabbitMQ Best Practices." https://www.cloudamqp.com/blog/part1-rabbitmq-best-practice.html, Sept. 2019.

[42] OpenTelemetry, "OpenTelemetry Documentation." https://opentelemetry.io/docs/, 2022.

[43] U. Zorrilla Castro, ".NET Core Apps: How to Debug and Diagnose." https://www.youtube.com/watch?v=Hw__DetUlOY, June 2021.

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

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

A.5 Anexo: Docker compose

En el fragmento A.1 mostramos la declaración de la configuración de un servicio. En las líneas 4

Podemos apreciar que nos permite declarar las dependencias entre servicios (líneas 12-13). Esto fue clave para el despliegue del contenedor de `RabbitMQ`. Debido a que el protocolo requiere de una conexión permanente al bus[41], todos los servicios que dependen de él deben desplegarse después. Por ello, declaramos una dependencia con este servicio y definimos una política de reintentos (línea 18).

```
1 climatisation_rules:
2 build:
3   context: ./climatisation_rules
4   args:
5     ANALYSISSERVICE_SERVICEURI: "http://analysis:80"
```

```
    BUSCONFIGURATION_SERVICEURI: "amqp://user:password@rabbitmq"
    GRAFANA_LOKI_URI: "http://loki:3100"
    JAEGER_HOST: "jaeger"
    JAEGER_PORT: "6831"
environment:
  ASPNETCORE_ENVIRONMENT: "Production"
depends_on:
  - rabbitmq
ports:
  - "9001:80"
networks:
  - analysis-network
restart: on-failure:5
```

Fragmento A.1: Ejemplo de declaración de despliegue de un servicio en Docker Compose