



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Escola Tècnica
Superior d'Enginyeria
Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

**Ingeniería y Diseño de soluciones de microservicios
auto-adaptativos.
Una aplicación práctica de los bucles de control sobre
*Kubernetes.***

Trabajo Fin de Máster

Máster Universitario en Ingeniería Informática

Autor: Miguel Blanco Máñez

Tutor: Joan Fons i Cors

Cotutor: Vicente Pelechano Ferragud

Curso : 2020-2021

*"La diferencia entre la teoría y la práctica es que, en teoría,
no hay diferencia entre la teoría y la práctica"*
– Benjamin Brewster, "The Yale Literary Magazine", Febrero 1882.



Resumen

Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde dicho software está compuesto por pequeños servicios independientes que se comunican a través de APIs bien definidas. Estas arquitecturas suelen ser desplegadas y gestionadas en plataformas que 'orquestan' estas soluciones a través de la infraestructura que proporcionan. En este sentido, *Kubernetes* se postula como una de las apuestas más sólidas en estos momentos como orquestador de microservicios .

En este trabajo se pretende dar un paso adelante y extender las capacidades de computación autónoma, no sólo a la infraestructura computacional – *Kubernetes* –, sino también a nivel de la propia arquitectura de microservicios. Para ello, se utilizarán técnicas provenientes de los bucles de control – en concreto, los bucles de control MAPE-K propuestos por IBM – que nos permitirá realizar esta ingeniería y diseño de este tipo de arquitecturas, en el que el diseño arquitectónico de las soluciones basadas en microservicios podrá adaptarse – en tiempo de ejecución – a las circunstancias, de manera autónoma, por la propia solución.

Se realizará un trabajo práctico en el que aplicando las técnicas de computación autónoma desarrolladas en el grupo Tatami con el enfoque *FADA* (Framework de auto-ADaptación basado en los bucles de control MAPE-K), se diseñará una solución auto-adaptativa que pueda aplicarse sobre soluciones de microservicios orquestadas con *Kubernetes*. Se propondrá una estrategia que permita, no sólo configurar en un momento determinado una solución de microservicios auto-adaptativos sobre *Kubernetes*, sino que también definirá operaciones de auto-gestión de la propia plataforma cuando la solución de microservicios se reconfigure de manera autónoma.

Para ejemplificar y validar el trabajo, se desarrollará un ejemplo prototípico real, en el que se demostrará la aplicación de la propuesta aplicando *FADA* sobre una solución de microservicios operando en clústers computacionales orquestados con *Kubernetes* y monitorizados con componentes software propios.

Palabras clave: *Kubernetes*, MAPE-K, microservicios, auto-adaptativo, arquitecturas.



Abstract

Microservices are an architectural and organizational approach to software development where such software is made up of small independent services that communicate through well-defined APIs. These architectures are usually deployed and managed on platforms that 'orchestrate' these solutions through the infrastructure they provide. In this sense, *Kubernetes* is postulated as one of the strongest bets at the moment as a microservices orchestrator.

This work aims to take a step forward and extend autonomous computing capabilities, not only to the computational infrastructure - *Kubernetes* - but also at the level of the microservices architecture itself. To do this, techniques from control loops will be used - specifically, the MAPE-K control loops proposed by IBM - which will allow us to carry out this engineering and design of this type of architecture, in which the architectural design of the solutions based on microservices can be adapted - at runtime - to circumstances, autonomously, by the solution itself.

A practical work will be carried out in which, applying the autonomous computing techniques developed in the Tatami group with the *FADA* approach (Framework of self-ADaptation based on MAPE-K control loops), a self-adaptive solution that can be applied will be designed on microservices solutions orchestrated with *Kubernetes*. A strategy will be proposed that allows, not only to configure at a certain moment a self-adaptive microservices solution on *Kubernetes*, but will also define self-management operations of the platform itself when the microservices solution is reconfigured autonomously.

To exemplify and validate the work, a real prototypical example will be developed, in which the application of the proposal will be demonstrated by applying *FADA* on a microservices solution operating in computational clusters orchestrated with *Kubernetes* and monitored with its own software components.

Keywords : *Kubernetes*, MAPE-K, microservices, self-adaptive, architectures.

Resum

Els microserveis són un enfocament arquitectònic i organitzatiu per al desenvolupament de programari on aquest programari està compost per petits serveis independents que es comuniquen a través d'APIs ben definides. Aquestes arquitectures solen ser desplegades i gestionades en plataformes que 'orquestrin' aquestes solucions a través de la infraestructura que proporcionen. En aquest sentit, *Kubernetes* es postula com una de les apostes més sòlides en aquests moments com orquestrador de microserveis.

En aquest treball es pretén fer un pas endavant i estendre les capacitats de computació autònoma, no només a la infraestructura computacional - *Kubernetes* -, sinó també a nivell de la pròpia arquitectura de microserveis. Per a això, s'utilitzaran tècniques provinents dels bucles de control - en concret, els bucles de control MAPE-K proposats per IBM - que ens permetrà realitzar aquesta enginyeria i disseny d'aquest tipus d'arquitectures, en el qual el disseny arquitectònic de les solucions basades en microserveis podrà adaptar - en temps d'execució - a les circumstàncies, de manera autònoma, per la mateixa solució.

Es realitzarà un treball pràctic en què aplicant les tècniques de computació autònoma desenvolupades en el grup Tatami amb l'enfocament *FADA* (Framework d'auto-adaptació basat en els bucles de control MAPE-K), es dissenyarà una solució acte-adaptativa que pugui aplicar-se sobre solucions de microserveis orquestrades amb *Kubernetes*. Es proposarà una estratègia que permeti, no només configurar en un moment determinat una solució de microserveis acte-adaptatius sobre *Kubernetes*, sinó que també definirà operacions d'auto-gestió de la pròpia plataforma quan la solució de microserveis es reconfigure de manera autònoma.

Per exemplificar i validar el treball, es desenvoluparà un exemple prototípic real, en el qual es demostrarà l'aplicació de la proposta aplicant *FADA* sobre una solució de microserveis operant en clústers computacionals orquestrats amb *Kubernetes* i monitoritzats amb components programari propis.

Paraules clau: *Kubernetes*, MAPE-K, microserveis, auto-adaptatiu, arquitectures.



Tabla de contenidos

1. Introducción	11
1.1 Motivación.....	12
1.2 Objetivos.....	13
1.3 Metodología y plan de trabajo.....	13
2. Contexto tecnológico	17
2.1 Herramientas.....	17
2.1.1 <i>Spring Boot</i>	17
2.1.2 <i>Minikube</i>	18
2.1.3 <i>Docker</i>	19
2.1.4 <i>Docker registry</i>	19
2.2 Entornos de desarrollo.....	20
2.2.1 IntelliJ.....	20
2.2.2 Postman.....	20
2.2.3 MQTT.fx.....	20
2.3 Notación para arquitecturas de microservicios.....	21
3. Caso de estudio.....	25
3.1 Introducción teórica a los bucles de control	25
3.1.1 MAPE-K.....	25
3.1.1.1 Monitor.....	26
3.1.1.2 Analizador	26
3.1.1.3 Planificador.....	26
3.1.1.4 Ejecutor.....	26
3.1.1.5 Base de conocimiento K.....	27
3.2 Patrón de arquitectura de microservicios.....	28
3.3 <i>Adaptive-ready service ARS/ARM</i>	29
3.4 <i>Kubernetes</i>	31
3.4.1 Objetos en <i>Kubernetes</i>	33
3.4.2 Archivos de creación de objetos.....	35
3.4.3 <i>Kubernetes</i> CLI, <code>kubect l</code>	36
3.5 Embalpack.....	36
4. Análisis del problema.....	39
4.1 Servicios <i>adaptive-ready ARS/ARM</i>	40
4.1.2 Requisitos. Contrato de Adaptación.....	40
4.2 <i>Kubernetes</i>	41
4.2.1 Requisitos.....	41

4.3 <i>Efactor</i>	42
4.3.1 Requisitos.....	42
4.4 <i>Sonda</i>	43
4.4.1 Requisitos.....	43
4.5 Registro de contenedores.....	43
5. Diseño de la solución.....	45
5.1 Servicio <i>adaptive-ready</i>	45
5.1.1 Operaciones <i>adaptive-ready</i>	45
5.1.2 Definición de API REST.....	47
5.2 <i>Efactor</i>	47
5.2.1 Funciones del <i>Efactor</i>	48
5.2.2 Definición de API REST.....	49
5.2.3 Traducción de mensajes.....	50
5.3 <i>Sonda</i>	50
5.3.1 Funciones de la <i>Sonda</i>	50
5.3.2 Definición de API REST.....	51
5.3.3 Comunicación con <i>FADA</i>	52
5.3.3.1 Definición de mensajes.....	52
5.4 Resumen del diseño.....	52
5.4.1 Inicio.....	53
5.4.2 Despliegue en <i>Kubernetes</i>	53
5.4.3 Operaciones sobre ARS/ARM.....	54
5.4.4 Operaciones auto-adaptativas.....	54
6. Implementación.....	55
6.1 Despliegue inicial en <i>Minikube</i>	55
6.1.1 Implementación de ARS/ARM.....	55
6.1.2 Creación de la imagen <i>Docker</i>	58
6.1.3 Archivos de creación de objetos.....	59
6.1.3.1 <i>Deployment</i>	59
6.1.3.2 <i>Service</i>	60
6.1.3.3 <i>Secret</i>	61
6.1.4 Ejecución del despliegue inicial.....	62
6.2 Implementación del <i>Efactor</i> y la <i>Sonda</i>	65
6.2.1 Implementación del <i>Efactor</i>	65
6.2.2 Implementación de la <i>Sonda</i>	68
6.2.3 Test del <i>Efactor</i> y la <i>Sonda</i>	72
6.2.3.1 Test del <i>Efactor</i>	72



6.2.3.1 Test de la <i>Sonda</i>	75
6.3 Implementación del caso de estudio. Embalpack	77
6.3.1 Componentes participantes.....	77
6.3.1.1 Semáforos de producción. DevTL.....	77
6.3.1.2 Máquinas de producción. DevPL.....	78
6.3.1.3 Extensión del sistema productivo. PL y TLC.....	78
6.3.1.4 Servicios de infraestructura. ERP de producción.....	79
6.3.1.5 Servicio auxiliar. TLCoffline.....	79
6.3.2 Escenario.....	80
6.3.2.1 Configuración arquitectónica.....	80
6.3.2.2 Escenario de adaptación.....	81
6.4 Test final.....	83
6.4.1 Resultados.....	84
7. Conclusiones.....	89
7.1 Impacto del trabajo realizado.....	90
7.1.1 Colaboración con artículo científico.....	90
7.1.2 Aplicación industrial.....	91
8. Referencias.....	92
Anexo	93
1. Cómo accede <i>Minikube</i> al repositorio privado	93
2. <i>Prometheus</i> y <i>Alertmanager</i>	95
3. Creación de un clúster <i>Kubernetes</i>	98

1. Introducción

Todo el software necesita una arquitectura para una mejor comprensión, comunicación, consenso y negociación de las diversas partes que lo integran. La arquitectura del software es la base de todo proyecto. Esta determina la seguridad, las dependencias, la orientación y la implementación de las pautas establecidas. Los microservicios y las monolíticas son arquitecturas de software populares y potentes que utilizan la mayoría, si no todos los ingenieros de software en la actualidad. El debate sobre funcionalidad y eficiencia continua entre los dos. Si bien algunas empresas están adoptando la nueva arquitectura de microservicios, otras aún mantienen la contra-parte anterior, monolítica (*fig. 1*).

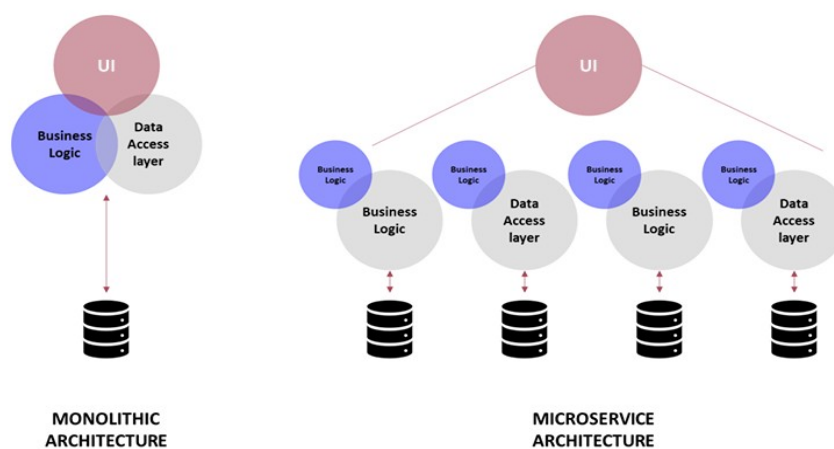


Figura 1. Esquema de la arquitectura monolítica y la basada en microservicios.

La arquitectura de aplicaciones monolítica es un enfoque en el que todos los componentes de una aplicación están conectados en una sola unidad. No hay módulos o servicios aislados, bases de código independientes o bases de datos independientes. El despliegue en la aplicación monolítica generalmente varía según cada aplicación pero suele ser muy sencillo, a veces incluso con un sólo punto de ejecución del mismo. La gestión de la aplicación monolítica también es muy sencilla en comparación con la arquitectura de microservicios. Para una aplicación muy pequeña, el enfoque monolítico parece funcionar bien a menos que la aplicación crezca a gran escala.

Como decimos, de lo que se le suele acusar a los sistemas monolíticos es la baja escalabilidad y la falta de flexibilidad a los cambios. Por ejemplo, ajustar el sistema al nuevo formato de los documentos contables puede llevar muchas semanas y agregar alguna función nueva, muchos meses. En el caso de la arquitectura de microservicios, es mucho más fácil y rápido, simplemente cambiamos un microservicio o añadimos uno nuevo, eliminando otro que ya no cumple su función.

Así pues los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de aplicaciones software donde esta está compuesta por pequeños servicios independientes que se comunican a través de APIs bien definidas. Este tipo de

arquitecturas se están imponiendo como un paradigma de computación sólido gracias a las facilidades de desarrollo, mantenimiento y escalado dinámico automático.

Con el avance de las infraestructuras en comunicación y en hardware, cada vez más la computación está migrando a lo que se llama ‘la nube’ – *cloud-computing* – proporcionando a las empresas soluciones en las que poder implementar sus sistemas en servidores, despreocupándose así de la actualización de los equipos, ya que sólo hay que actualizar y mantener el servidor, entre otras ventajas. Las soluciones basadas en microservicios suelen operar típicamente sobre plataformas en la nube, que montan estas soluciones a través de una infraestructura que permite monitorizar y desplegar/replegar microservicios dinámicamente. En este sentido, *Kubernetes* se postula como una de las apuestas más sólidas en estos momentos como orquestador de microservicios – y contenedores – .

Desarrollando componentes o utilizando herramientas para la sensorización y monitorización del estado de dichos microservicios conseguimos – junto a *Kubernetes* – una combinación que proporciona una plataforma computacional en la nube con ciertas capacidades de auto-adaptación, en la que se consigue que los microservicios estén siempre disponibles y con un nivel de carga adecuado, a través de operaciones de escalado y desescalado dinámico.

1.1 Motivación

Observando la arquitectura de microservicios y la utilización de orquestadores de contenedores nos hacemos la siguiente pregunta, ¿y si pudiéramos crear un sistema auto-adaptativo basado en esta arquitectura y tecnología?

Actualmente los orquestadores si bien ofrecen una gestión automatizada de recursos no están diseñados para gestionar automáticamente cambios arquitectónicos sobre la configuración de servicios, necesarios bajo ciertas condiciones operativas.

Una de las soluciones es utilizar bucles de control. El aspecto desafiante de diseñar e implementar un sistema auto-adaptativo es que no sólo el sistema debe aplicar cambios en el tiempo de ejecución, sino que también deben cumplir con los requisitos del sistema hasta un nivel satisfactorio [1][2][3].

Si aplicamos la computación autónoma – a través del uso de estos bucles – como estrategia para la reconfiguración dinámica de arquitecturas de microservicios podemos introducir el concepto de servicio *adaptive-ready*, como un servicio que ofrece un contrato de adaptación diseñado para ser usado por los bucles de control.

FADA (Framework de auto-ADaptación) es el conjunto de *frameworks* de adaptación que está siendo desarrollado por el grupo TaTami del Centro de Investigación PROS que proporciona marcos para el desarrollo de software con capacidades de computación autónoma mediante bucles de control, aplicando soluciones de modelado y generación de código y usando modelos en tiempo de ejecución – *models-at-runtime* – . Estos

sistemas se diseñan de manera que tanto los modelos del sistema, como los componentes que lo implementan se desarrollan a la par usando un *framework* de implementación – que propone el propio método – para construir estos sistemas auto-adaptativos.

Si combinamos estos tres elementos, *Kubernetes* como orquestador, *FADA* como bucle de control y servicios *adaptive-ready*, seremos capaces de crear un sistema auto-adaptativo que trabaje en la nube.

1.2 Objetivos

Así pues, nuestro objetivo será diseñar e implementar un conjunto de servicios *adaptive-ready* y desplegar una arquitectura de microservicios reconfigurable sobre *Kubernetes* usando *FADA* como bucle de control.

Para ejemplificar la propuesta se utiliza un caso de estudio real en el que se aplican estos principios para desarrollar el sistema informático de una fábrica de producción industrial.

Utilizaremos *Minikube*, que es una distribución reducida de *Kubernetes* que permite alcanzar el máximo rendimiento funcional de esta herramienta de una manera intuitiva. El orquestador gestionará contenedores creados con *Docker*, que se alojarán en un repositorio privado – *Docker registry* –, el cuál también crearemos.

Así mismo se van a desarrollar e implementar varios componentes *adaptive-ready* que serán desplegados/replegados según el estado actual del sistema y su configuración.

Para completar el escenario se diseñará e implementará un *Efector* y una *Sonda*, que serán una extensión de la herramienta *FADA* y que servirá de punto de unión entre el bucle de control y *Kubernetes*.

1.3 Metodología y plan de trabajo

A continuación vamos a describir nuestro plan de trabajo previsto.

Primero realizaremos el diseño e implementación de un conjunto de servicios *adaptive-ready*, que serán los participantes en nuestro escenario.

Seguiremos con la ‘contenerización’ de estos servicios utilizando *Docker* y crearemos el repositorio privado para imágenes *Docker*, *Docker registry*.

Una vez tengamos nuestro repositorio pasaremos a la creación de un clúster *Kubernetes* usando *Minikube*. Daremos acceso a *Minikube* para que pueda descargar las imágenes de nuestro repositorio privado.

Una vez tengamos la comunicación abierta pasaremos al despliegue inicial de nuestro escenario todavía sin automatizar.



Seguidamente realizaremos el diseño e implementación de nuestro *Efactor* que será el encargado de ejecutar las órdenes que son enviadas por el bucle de control. Estas órdenes las ejecutará directamente sobre *Minikube* y los servicios *adaptive-ready*.

Para finalizar con la parte de diseño e implementación crearemos una *Sonda* que se encargará de la recopilación del estado de los dispositivos virtuales. Esta *Sonda* reportará sus datos al monitor del bucle, el cuál analizará los datos y actuará en consecuencia.

Por último conectaremos las distintas partes con la herramienta *FADA* y procederemos a la implementación final. Se testará y mostrarán los resultados obtenidos.

En lo que concierne a este documento comenzaremos por mostrar y comentar las herramientas que vamos a utilizar para el desarrollo del trabajo. Realizaremos una breve descripción de las mismas, señalando la función que desempeñaran cada una dentro del proyecto.

Para poner el proyecto en perspectiva y poder determinar su contexto haremos una introducción teórica donde veremos los fundamentos de la computación autónoma basada en bucles de control y los servicios *adaptive-ready*. Describiremos el propósito y funcionamiento de *Kubernetes*. Una vez hayamos montado el escenario de trabajo pondremos en contexto el proyecto.

Llegados a este punto realizaremos el análisis del problema, describiendo donde se encuentra cada componente y la función que van a desempeñar en nuestro escenario.

Posteriormente presentaremos el diseño de nuestra solución donde enseñaremos su estructura, conexiones y comunicaciones, haciendo mención a las distintas evoluciones de la misma hasta llegar al diseño final. Para ello emplearemos la notación SAS [4].

Para continuar pasaremos a implementar la solución siguiendo el guión del diseño. La implementación se divide en tres fases, una primera donde describiremos los pasos seguidos para realizar un despliegue inicial en *Minikube*, la segunda donde implementaremos el *Efactor* y la *Sonda* y una última donde explicaremos la implementación final donde entra en juego el bucle MAPE-K.

Realizaremos nuestras conclusiones y finalmente comentaremos en el anexo aquellas cuestiones o problemas que han sido de especial relevancia, como son dar acceso a *Minikube* a nuestro repositorio privado. En este apartado comentaremos también los distintos “camino” – todos útiles – a los que nos ha llevado este proyecto, por ejemplo el uso de *Prometheus* y *Alertmanager* alojados en *Kubernetes* como *Sondas* en nuestro escenario o la creación de un clúster *Kubernetes* utilizando tres máquinas *Ubuntu*.

A continuación vamos a realizar una breve cronología que mostraremos en un diagrama de Gantt.

Tarea 1 (**T 1**) → Entrevista con el tutor y estudio del problema.

Tarea 2 (**T 2**) → Análisis de requisitos.

Tarea 3 (**T 3**) → Elección de herramientas.

Tarea 4 (**T 4**) → Diseño e implementación servicios *adaptive-ready*. *Docker*.

Tarea 5 (**T 5**) → Creación *Minikube*. Configurar permisos de acceso a *Docker*.

Tarea 6 (**T 6**) → Despliegue inicial. Test de funcionamiento. Depuración.

Tarea 7 (**T 7**) → Diseño e implementación del *Efactor* y la *Sonda*.

Tarea 8 (**T 8**) → Implementación final.

Tarea 9 (**T 9**) → Test de funcionamiento. Depuración.

Tarea 10 (**T 10**) → Documentación

Tarea 11 (**T 11**) → Entrega

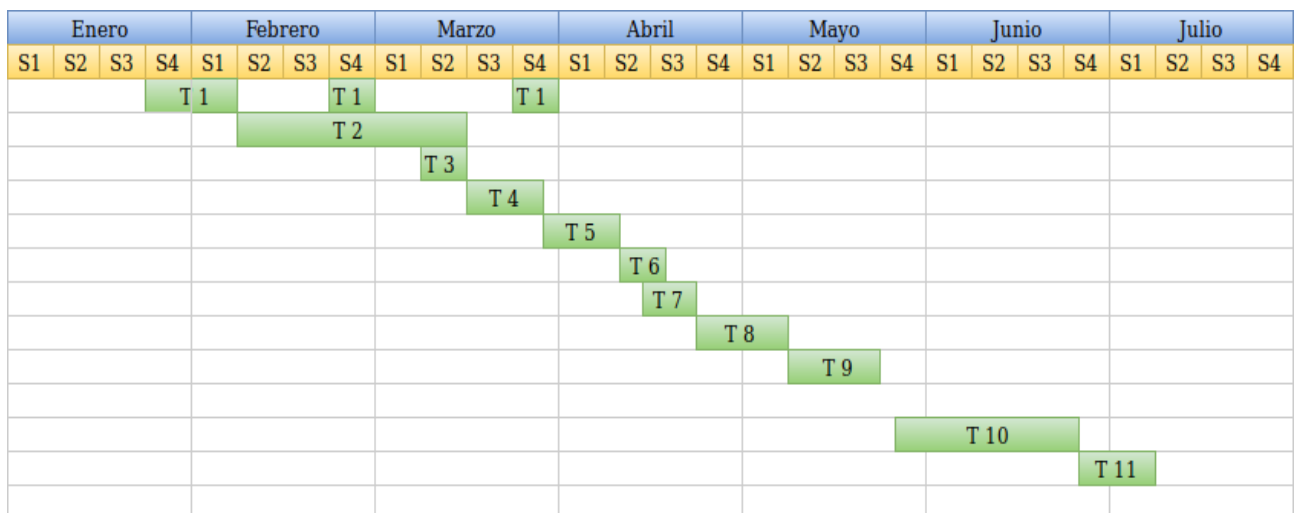


Diagrama de Gantt resultante.

2. Contexto tecnológico

En este capítulo vamos a mostrar las herramientas y entornos que emplearemos para el desarrollo del trabajo. Todo el software que se emplea es *open source software* o software de código abierto, es decir, aquel cuyo código fuente y otros derechos que normalmente son exclusivos para quienes poseen los derechos de autor, son publicados bajo una licencia de código abierto o forman parte del dominio público, con lo que no necesitamos de licencias del fabricante para su uso y además su descarga es gratuita.

Para el diseño e implementación de todos nuestros microservicios – *adaptive-ready*, *Efector* y *Sonda* – utilizaremos *Spring Boot*. Para el clúster *Kubernetes* será *Minikube* nuestra elección.

2.1 Herramientas

2.1.1 *Spring Boot*

Java Spring Framework (*Spring Framework*) es un popular marco de código abierto a nivel empresarial para crear aplicaciones independientes de nivel de producción que se ejecutan en la máquina virtual Java (JVM).

Spring Framework ofrece una función de inyección de dependencias que permite a los objetos definir sus propias dependencias que posteriormente el contenedor *Spring* inyecta en ellos. Esto permite a los desarrolladores crear aplicaciones modulares que constan de componentes poco acoplados que son ideales para microservicios y aplicaciones de red distribuidas.

Spring Framework también ofrece soporte integrado para tareas típicas que una aplicación necesita realizar, como enlace de datos, conversión de tipos, validación, manejo de excepciones, manejo de recursos y eventos, internacionalización y más. Se integra con varias tecnologías *Java EE* como *RMI* (Invocación de método remoto), *AMQP* (Protocolo de cola de mensajes avanzado), Servicios web *Java* y otros. En resumen, *Spring Framework* proporciona a los desarrolladores todas las herramientas y características que necesitan para crear aplicaciones *Java EE* multiplataforma poco acopladas que se ejecutan en cualquier entorno.

Java Spring Boot (*Spring Boot*) es una herramienta que hace que el desarrollo de aplicaciones web y microservicios con *Spring Framework* sea más rápido y fácil a través de tres capacidades principales:

- **Autoconfiguración:** Los desarrolladores pueden configurar automáticamente su aplicación *Spring*. Sin embargo, *Spring Boot* también es capaz de cambiar la configuración en función de las dependencias que enumere.



- Un enfoque orientado a la configuración: *Spring Boot* decide qué valores predeterminados usar para la configuración. Además, decide qué paquetes instalar para las dependencias que necesita. Esto ayuda a los desarrolladores a comenzar rápidamente en sus proyectos.
- La capacidad de crear aplicaciones independientes: No es necesario implementar la aplicación en un servidor web. Simplemente hay que ejecutar el programa para iniciar la aplicación.

Estas características funcionan juntas para brindar una herramienta que permite configurar una aplicación basada en *Spring* con una configuración mínima.

En este trabajo utilizaremos *Spring Boot* para la implementación de los microservicios.

2.1.2 Minikube

Minikube es la solución ideal para pequeños proyectos basados en contenedores. Permite, por ejemplo, configurar un clúster de *Kubernetes* en privado sin tener que trabajar directamente con todo un servidor o una nube. Al contrario que *Kubernetes*, *Minikube* prescinde de grandes infraestructuras y puede configurar clústeres fácilmente de forma local.

Un ordenador y un clúster con un solo nodo: este es el espacio que necesita *Minikube*. Este requisito mínimo es especialmente adecuado para pequeños proyectos privados, que los desarrolladores de software ya pueden implementar fácilmente gracias a *Minikube*. No hace falta un servidor ni una nube, ya que el clúster de *Kubernetes* simplemente se ejecuta en el localhost. *Minikube* funciona de forma predeterminada con VirtualBox como software de virtualización, por lo que no solo se puede utilizar en Linux, sino también en Windows o macOS. Sin embargo, si se prefiere trabajar sin VirtualBox, *Minikube* también consigue expandirse de manera que no requiera este software.

La simplicidad de *Minikube* también se nota en sus funciones. La herramienta es compatible con las siguientes funciones de *Kubernetes*:

- DNS
- NodePorts
- ConfigMaps y Secrets
- Dashboards
- Entornos de tiempo de ejecución de contenedores: *Docker* o alternativas a *Docker* como rkt, CRI-O o containerd.
- Compatibilidad con CNI (Container Network Interface)
- Ingress

2.1.3 Docker

Un contenedor es una unidad estándar de software que empaqueta el código y todas sus dependencias para que la aplicación se ejecute de forma rápida y confiable de un entorno informático a otro. Una imagen de contenedor de *Docker* es un paquete de software ligero, independiente y ejecutable que incluye todo lo necesario para ejecutar una aplicación: código, tiempo de ejecución, herramientas del sistema, bibliotecas del sistema y configuraciones.

Las imágenes de contenedor se convierten en contenedores en tiempo de ejecución y, en el caso de los contenedores de *Docker*, las imágenes se convierten en contenedores cuando se ejecutan en *Docker Engine*. Disponible para aplicaciones basadas en Linux y Windows, el software en contenedores siempre se ejecutará de la misma manera, independientemente de la infraestructura.

La tecnología de contenedores *Docker* se lanzó en 2013 como un motor *Docker* de código abierto. Aprovechó los conceptos informáticos existentes en torno a los contenedores y, específicamente, en el mundo de Linux, primitivas conocidas como cgroups y namespaces. La tecnología de *Docker* es única porque se centra en los requisitos de los desarrolladores y operadores de sistemas para separar las dependencias de las aplicaciones de la infraestructura.

En nuestro proyecto los microservicios creados con *Spring* serán convertidos en imágenes de contenedor utilizando *Docker*.

2.1.4 Docker registry

Un registro de contenedores es un espacio central sin estado y altamente escalable para almacenar y distribuir imágenes de contenedores. Proporcionan una gestión de imágenes segura y una forma rápida de extraer y enviar imágenes con los permisos adecuados.

El registro de contenedores más conocido es *DockerHub*, que es el registro estándar para *Docker* y *Kubernetes*. El único problema con los registros públicos es que no se tiene control total sobre sus acciones y pueden resultar costosas si se necesitan varias imágenes privadas. Además si se están utilizando imágenes que forman parte de un proyecto en desarrollo, la confidencialidad de ese material debe ser máxima, si utilizamos un repositorio público estamos exponiendo nuestro trabajo a cualquiera. Es por eso que alojar un registro privado propio puede resultar útil en muchos casos.

Los registros privados brindan múltiples opciones de autenticación y almacenamiento diferentes y se pueden personalizar según requisitos individuales. *Docker* proporciona una imagen oficial llamada *registry* con la cual se puede crear un registro de contenedores privado.

Utilizaremos esta imagen y la funcionalidad que nos proporciona el software que contiene para la creación de nuestro repositorio privado.



2.2 Entornos de desarrollo

Seguidamente enumeraremos los entornos de desarrollo que vamos a utilizar y que nos ayudarán a generar código en unos casos y a la realización de tests en otros.

2.2.1 IntelliJ IDEA

Este IDE forma parte de los entornos desarrollados por *JetBrains*, el máximo representante de esta compañía es el archiconocido *Android Studio*. Es una de las mejores alternativas al uso de *Eclipse IDE* para el desarrollo en *Java*. Este IDE puede generar proyectos tanto para *Maven*, como para *Gradle* o *Kotlin*.

En este trabajo hemos creado proyectos utilizando *Gradle* y *Maven*, y este IDE resulta muy intuitivo a la hora de trabajar con librerías o dependencias en nuestros proyectos.

2.2.2 Postman

Postman es una herramienta que se utiliza, sobre todo, para el testing de API REST, aunque también admite otras funcionalidades que se salen de lo que engloba el testing de este tipo de sistemas. Ha alcanzado una popularidad tal que es casi una herramienta estándar para desarrolladores Web.

La función principal de Postman es, sencillamente, realizar peticiones HTTP (GET, POST, DELETE, UPDATE...) a una dirección de nuestro interés. Esto es de gran utilidad a la hora de interactuar con APIs Web e, incluso, para testear desarrollos propios.

Postman nos permite modificar todos los parámetros de la petición. Así, *podemos* editar el header de la petición. También *podemos* editar el contenido, y añadir campos tanto en Json, XML o texto plano, entre otros.

Utilizaremos Postman para testar nuestros microservicios – *adaptive-ready*, *Efector* y *Sonda* – en todas sus fases de desarrollo. El objetivo es comprobar las API REST implementadas y asegurarnos de su correcto funcionamiento antes de ponerlas en producción.

2.2.3 MQTT.fx

MQTT.fx es un cliente MQTT escrito en Java basado en *Eclipse Paho*. El programa proporciona perfiles de conexión para conectarse a diferentes *brokers*. Los perfiles de conexión permiten la configuración de opciones de conexión como identificación de cliente, SSL / TLS, nombre de usuario / contraseña y última voluntad y testamento.

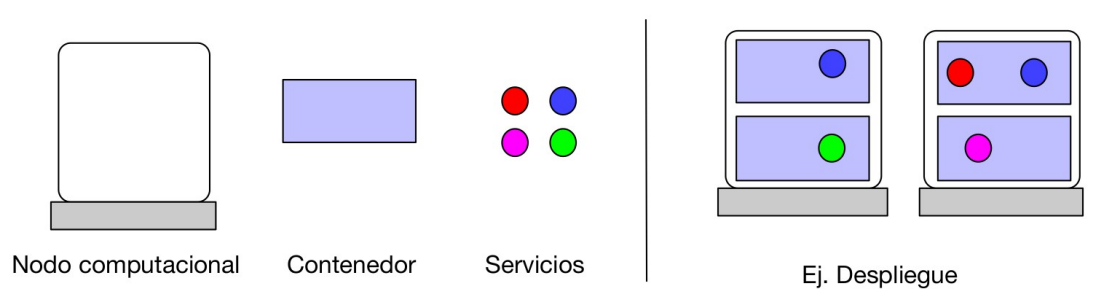
Utilizaremos MQTT.fx para probar la *Sonda* que deberá reportar los datos al monitor a través de colas de mensajería.

2.3 Notación para arquitecturas de microservicios

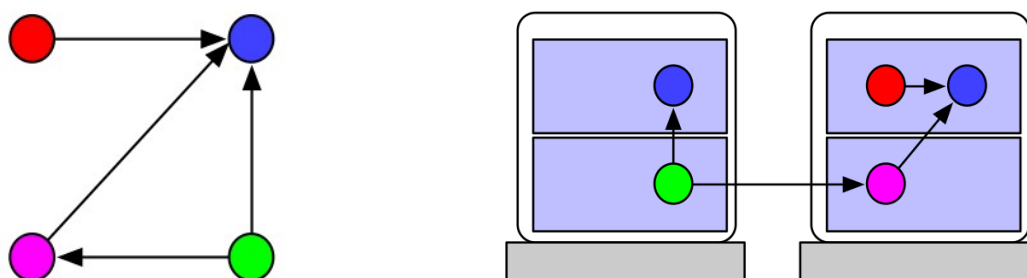
Para ilustrar el contenido del texto se va a utilizar una notación propuesta por J.Fons, en el documento “Arquitecturas basadas en microservicios. Conceptualización y Notación. Framework, Prototipo y Ejemplos”.

Las figuras básicas son:

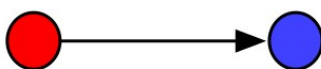
Nodo computacional, contenedor y servicios:



Vínculos o *bindings*:

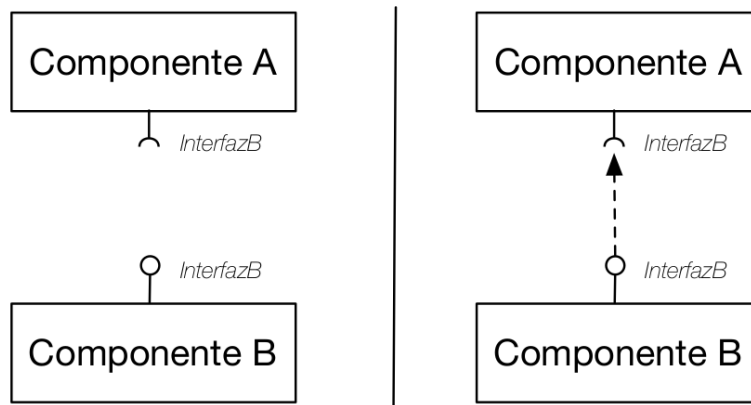


Para este trabajo se van a utilizar un tipo de *endpoint* que es el que define el propio microservicios al ofrecer funcionalidad y al requerirla. El vínculo básico:



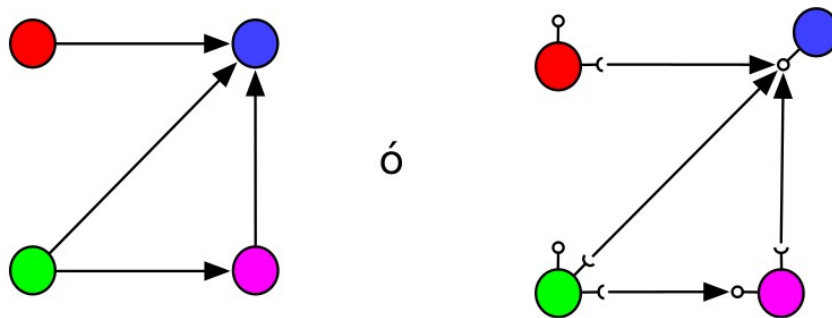
Ocurre porque un microservicio (rojo) requiere la funcionalidad que ofrece otro microservicio (azul). Para atender a esta necesidad, se propone utilizar la notación proveniente de diagramas de componentes donde un componente 'requiere' un servicio

que ofrece otro componente a través de una interfaz y que se refleja de la siguiente manera:



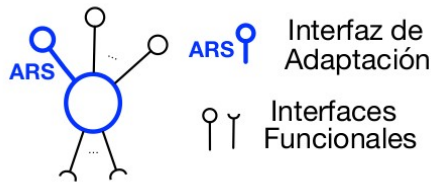
En esta notación (izquierda de la figura), el componente A requiere un servicio (que cumple interfaz 'InterfazB') que el componente B provee. Esto se describe en tiempo de diseño de los componentes. En tiempo de ejecución, si se quiere que el componente A esté conectado al servicio InterfazB del componente B, es necesario establecer un *bind*, o vínculo (flecha discontinua, a la derecha de la figura).

Siguiendo este razonamiento, aplicado a los microservicios tendremos lo siguiente:

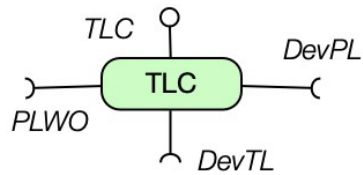


Desde el punto de vista de los *endpoints* antes comentados, además de definir los necesarios para comunicar contenedores y nodos computacionales, definiremos otros 'especiales' ligados a los microservicios: los *endpoints* de tipo 'service supply' y 'service require' de los microservicios.

Así pues definiremos los *adaptive-ready service ARS* de la siguiente manera:



Definiremos una interfaz de adaptación de tipo 'service supply' y las interfaces funcionales de los microservicios que internamente serán como se muestra:



Donde de igual manera ofrecerá interfaces del tipo 'service supply' (⊖) y 'service require' (⊕).

3. Caso de estudio

3.1 Introducción teórica al bucle de control

Un sistema de computación autónoma es un sistema que detecta su entorno operativo, modela su comportamiento en ese entorno y toma medidas para adaptarse a este. Un sistema informático autónomo tiene las propiedades de auto-configuración, auto-corrección, auto-optimización y auto-protección [1][2].

Para cumplir estas propiedades, se plantea la figura del *autonomic manager* (administrador autónomo) que es un componente que administra otros componentes de software o hardware mediante un bucle de control (*fig. 2*) y que se encuentra por encima de los sistemas que no son auto-adaptativos (o autónomos). El bucle de control del *autonomic manager* incluye funciones de monitorización, análisis, planificación y ejecución .

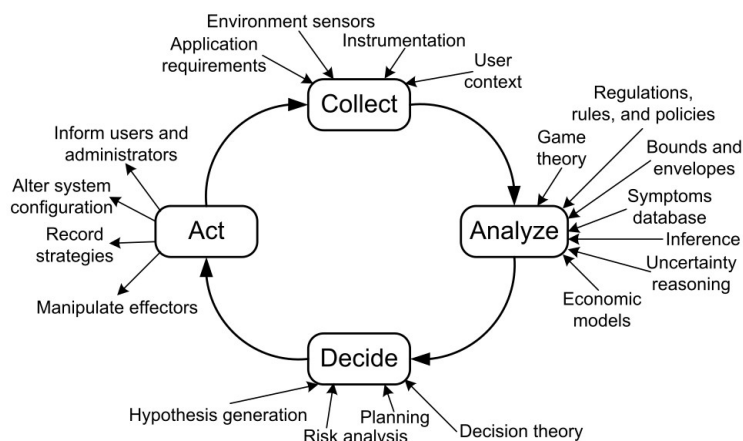


Figura 2. Bucles de control en un autonomic manager.

3.1.1 MAPE-K

Un gran avance en hacer explícitos los bucles de control vino en 2005 con la iniciativa de computación autónoma de IBM mostrando su énfasis en los sistemas de autogestión. Uno de los hallazgos clave de esta iniciativa de investigación es el plan de acción para construir sistemas autónomos utilizando *MAPE-K control loops* [1][2][3][5]. Tiene este nombre por las fases principales: Monitor, Analizador, Planificador y Ejecutor, la K viene de Knowledge – conocimiento – .

A continuación haremos una breve descripción de los componentes:

3.1.1.1 Monitor

La función de monitorización recopila los detalles de los recursos administrados, a través conectores o sensores, y los relaciona con los datos que se pueden analizar. Los detalles pueden incluir información de topología, métricas, configuraciones de componentes, etc. Estos datos incluyen información sobre la configuración de los recursos administrados, el estado, la capacidad ofrecida y el rendimiento.

Algunos de los datos son estáticos o cambian lentamente, mientras que otros son dinámicos y cambian continuamente a lo largo del tiempo. La función de monitorización agrega, correlaciona y filtra estos detalles hasta que determina un conjunto de datos que debe analizarse.

En nuestro proyecto el componente *Sonda* reportará sus mediciones a este Monitor.

3.1.1.2 Analizador

La función de análisis proporciona los mecanismos para observar y analizar situaciones para determinar si es necesario realizar algún cambio. Por ejemplo, el requisito para ejecutar un cambio puede ocurrir cuando la función de análisis determina que no se cumple alguna política. La función de análisis es responsable de determinar si el *autonomic manager* puede cumplir con la política establecida, ahora y en el futuro. En muchos casos, la función de análisis modela el comportamiento complejo, por lo que puede emplear técnicas de predicción como el pronóstico de series de tiempo y los modelos de colas. Estos mecanismos permiten al *autonomic manager* aprender sobre el entorno operativo y le ayudan a predecir el comportamiento futuro.

Los *autonomic manager* deben poder realizar análisis de datos complejos y razonar sobre las incidencias proporcionadas por la función de monitor. Si se requieren cambios, la función de análisis genera una solicitud de cambio y, lógicamente, pasa esa solicitud de cambio a la función de plan. La solicitud de cambio describe las modificaciones que el componente de análisis considera necesarias o convenientes.

3.1.1.3 Planificador

La función de plan crea o selecciona un procedimiento para ejecutar una alteración deseada en el recurso administrado. La función de plan puede tomar muchas formas, desde un solo comando hasta un flujo de trabajo complejo.

3.1.1.4 Ejecutor

La función de ejecución proporciona el mecanismo para programar y realizar los cambios necesarios en el sistema. Una vez que un *autonomic manager* ha generado un plan de cambio que corresponde a una solicitud de cambio, es posible que se deban tomar algunas medidas para modificar el estado de uno o más recursos administrados. La función de ejecución de un *autonomic manager* es responsable de llevar a cabo el procedimiento generado por la función de plan a través de una serie de acciones. Parte

de la ejecución del plan de cambios *podría* incluir la actualización de los conocimientos (K) que utiliza el *autonomic manager*.

Esta función es interesante ya que será la responsable de la comunicación con el *Efector* que ejecutará las acciones sobre *Kubernetes* y los servicios *adaptive-ready*.

3.1.1.5 Base de conocimiento K

Una base de conocimiento K es una implementación de un registro, diccionario, base de datos u otro repositorio que proporciona acceso al conocimiento de acuerdo con las interfaces prescritas por la arquitectura. En un sistema autónomo, el conocimiento consiste en tipos particulares de datos con sintaxis y semántica arquitectónicas, tales como incidencias, políticas, solicitudes de cambio y planes de cambio. Estos datos se pueden almacenar en una base de conocimiento para que se pueda compartir entre los componentes del *autonomic manager*.

Los datos almacenados se pueden utilizar para ampliar las capacidades de conocimiento de un *autonomic manager*, el cual puede cargar datos de una o más bases de conocimiento, y el administrador del *autonomic manager* puede activar ese conocimiento, lo que permite realizar tareas de administración adicionales (como reconocer incidencias específicas o aplicar ciertas políticas) [1][5].

De las distintas formas para obtener la base de conocimiento, nuestro caso se produce cuando es el propio *autonomic manager* el que crea el conocimiento.

- El **conocimiento** utilizado por un *autonomic manager* en particular podría ser creado por la parte del **monitor**, según la información recopilada a través de sensores. La parte del monitor puede crear conocimiento basado en actividades recientes al registrar las notificaciones que recibe de un recurso administrado.
- La parte de **ejecución** de un *autonomic manager* puede actualizar el conocimiento para indicar las acciones que se tomaron como resultado del **análisis** y la **planificación** según los datos monitorizados, la parte de ejecución indicaría cómo esas acciones afectaron el recurso administrado – basado en los datos subsiguientes monitorizados –.

Para concluir diremos que nuestro trabajo se relacionará con el Monitor y el Ejecutor a través del *Efector* y la *Sonda* que implementaremos (*fig. 3*).



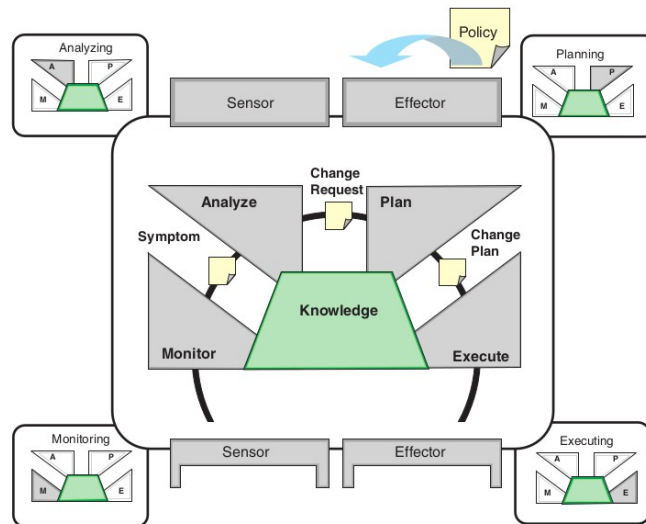


Figura 3. Esquema de la estructura interna de un autonomic manager que utiliza MAPE-K.

3.2 Patrón de Arquitectura de microservicios

El patrón de Arquitectura de microservicios tiene una serie de ventajas importantes [6][7]. En primer lugar, trata el problema de la complejidad y la adaptación a las nuevas tecnologías. Descompone lo que de otro modo sería una aplicación monolítica enorme en un conjunto de servicios. Si bien la cantidad total de funcionalidades es la misma, la aplicación se ha desglosado en un conjunto de servicios asequibles donde cada uno se ocupa de una única funcionalidad.

El patrón de Arquitectura de microservicios obliga a un nivel de modularidad que en la práctica es extremadamente difícil de conseguir con el código base de una típica aplicación monolítica. En consecuencia, los servicios individuales son mucho más rápidos para desarrollarse, y mucho más fáciles de entender y mantener.

Esta arquitectura también permite que cada servicio pueda ser desarrollado dependiendo de las necesidades de cada desarrollador, con el lenguaje, técnicas, métodos, etc. que él considere oportunas ya que, independientemente de la forma en que se desarrolle este debería poder comunicarse mediante una API. Esta libertad hace que los desarrolladores ya no estén obligados a utilizar las tecnologías, posiblemente obsoletas, que existían al inicio de un nuevo proyecto y puedan empezar a desarrollar el nuevo servicio con tecnologías más actuales. Además, como los servicios son relativamente pequeños, se hace posible reescribir un servicio antiguo para actualizarlo a una tecnología actual.

Los microservicios se pueden desplegar de manera independiente. Los desarrolladores no deben coordinar más el despliegue de cambios locales a su servicio. Este tipo de cambios se pueden implementar tan pronto como se hayan probado. Además, como ya hemos comentado antes, el patrón de Arquitectura de microservicios permite escalar de forma independiente cada servicio. Podemos implementar sólo el número de instancias de cada servicio que satisfacen sus restricciones de capacidad y disponibilidad. Además,

podemos utilizar los recursos que se adaptan mejor a los requisitos de recursos de un servicio.

Como cualquier tecnología, todas presentan ventajas e inconvenientes y esta no iba a ser la excepción. Un inconveniente importante de los microservicios es la complejidad que se desprende del hecho de que una aplicación de este tipo es un sistema distribuido. Los desarrolladores deben elegir e implementar un mecanismo de comunicación entre procesos basado en mensajería. Además, también tienen que escribir código para gestionar un error parcial ya que el destino de una solicitud puede ser lenta o no disponible. Por lo tanto el sistema de comunicaciones debe ser robusto para garantizar la consistencia de los datos y, en consecuencia, su desarrollo es más complejo que el de una aplicación monolítica.

Una aplicación de microservicios consiste generalmente en un gran número de servicios. Cada servicio tendrá varias instancias de ejecución. Se trata de muchas partes móviles que hay que configurar, desplegar, escalar y controlar. Además, los microservicios son dinámicos y se pueden desplegar en cualquier sistema, cambiar de ubicación a voluntad, replegarse o desplegarse de nuevo, así que hay que tener en cuenta que debemos controlar la ubicación y la disponibilidad en todo momento [8]. Por tanto, también será necesario implementar un mecanismo de descubrimiento de servicios que permita a un servicio descubrir las ubicaciones (host y puertos) de cualquier otro servicio con el que necesite comunicarse. En consecuencia, desplegar con éxito una aplicación de microservicios requiere un mayor control de los métodos de desarrollo por parte de los desarrolladores y un alto nivel de automatización.

3.3 Adaptive-ready service ARS/ARM

La disciplina de Ingeniería de Software se enfrenta al desafío de establecer las bases que permitan el desarrollo, despliegue, gestión y evolución sistemáticos de las futuras generaciones de sistemas auto-adaptativos – SAS – [9] .

Los sistemas auto-adaptativos son sistemas que pueden desarrollarse y aprender por sí solos y en los que tanto sus componentes, como su estructura se adaptan al ambiente en tiempo real. Es decir, pueden cambiar de forma autónoma su comportamiento como respuesta a los cambios tanto en los estados internos del sistema como en el entorno. Estos sistemas están generalmente asociados con sistemas que reciben gran cantidad de datos y el modo de operación es en tiempo real [3]. La auto-adaptabilidad puede ser estática o dinámica. En la auto-adaptación estática, los mecanismos de adaptación son definidos explícitamente por los diseñadores para que el sistema sepa cuál de estos elegir durante su ejecución, mientras que en la auto-adaptación dinámica, las adaptaciones las producen, planes de adaptación y requerimientos observados durante la monitorización. Estas adaptaciones son elegidas por el sistema en tiempo real.

El carácter propio de la adaptación asegura un nivel mínimo de supervisión humana, ya que el humano necesita tener un mínimo de control sobre el sistema y saber qué está

haciendo mientras trabaja de forma autónoma. Además, es necesario para que las aplicaciones que deban funcionar continuamente, incluso en condiciones adversas, sean capaces de cambiar los requerimientos siempre que sea necesario.

El paradigma de los SAS está basado en el término de evolucionar (expandir o reducir) la estructura de sistema para que esta pueda ser capaz de adaptarse a los cambios en el entorno. Esto puede suponer un reto ya que el sistema debe ser capaz de adaptarse a las situaciones dinámicamente y en tiempo real. Típicamente, un SAS consiste en un subsistema gestionado que implementa las funcionalidades del sistema y un subsistema gestor (autónomo) que implementa la lógica de adaptación (*fig. 3*).

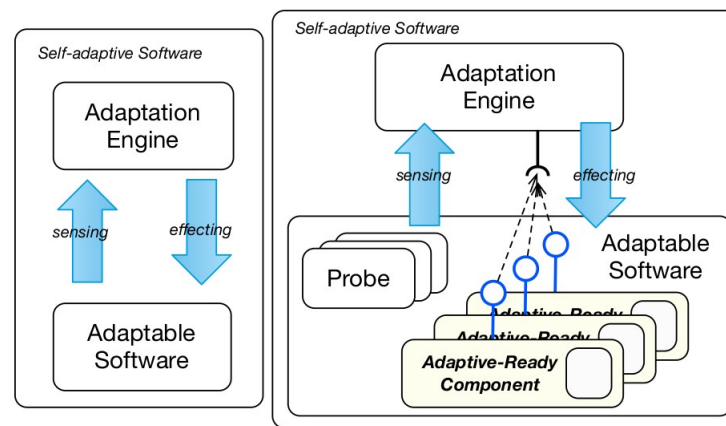


Figura 3. (Izq.) Esquema de software auto-adaptable; (Dcha.) Esquema de software auto-adaptable con componentes adaptive-ready.

Para diseñar correctamente un SAS, la comunidad informática propone un enfoque para diseñar y desarrollar SAS mediante el uso de circuitos de retroalimentación – *feedback loops* – como motores de adaptación en el nivel superior de los sistemas de software. Los esfuerzos en este área se han centrado en hacer más explícito el papel de los circuitos de retroalimentación. De esta manera, varias soluciones proporcionan estrategias globales para desarrollar estos circuitos de retroalimentación y sus componentes internos. Pero existen necesidades para un diseño completo, se necesitan técnicas para ayudar a los diseñadores y desarrolladores a diseñar los sistemas gestionados de modo que puedan estar listos para ser utilizados por los ciclos de retroalimentación. Así mismo se requieren marcos y *middleware*, proporcionando a los desarrolladores código generado automáticamente y modelos reutilizables para respaldar sus decisiones de diseño específicas [9].

Para responder a esta problemática, en el grupo Tatami del Centro PROS de la UPV introducen el concepto de ARC – *Adaptive Ready Component* –. Un ARC enriquece el concepto de componente de software proporcionando propiedades e interfaces específicas para facilitar los procesos de adaptación. Se pueden pues diseñar un sistema gestionado como facilitador esencial para implementar SAS a través de circuitos de retroalimentación [9] (*fig. 4*).

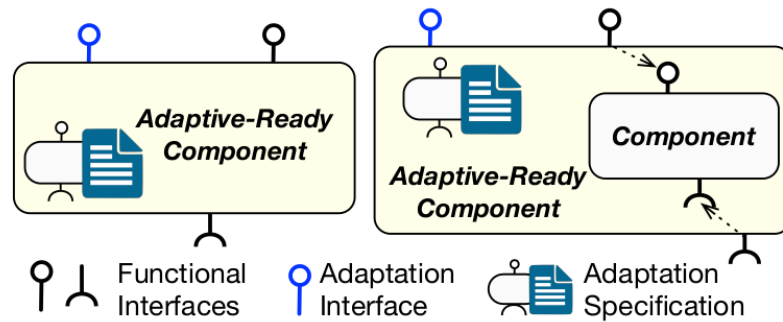


Figura 4. (Izq.) esquema ARC; (Dcha.) Esquema ARC para tratar con componentes heredados[9].

Bien, pues un *adaptive-ready service* – ARS – [11], es un componente software que hereda todas las características auto-adaptativas de ARC, pero que se enfoca en los servicios y sus características – comunicaciones, interfaces, disponibilidad, etc – . Este ARS encapsula al servicio para hacerlo auto-adaptativo.

En este trabajo también se utilizará la definición ARM que corresponde a *adaptive-ready microservice*, se trata solamente de un renombramiento para utilizarlo con microservicios.

3.4 Kubernetes

La responsabilidad principal de *Kubernetes* es la orquestación de contenedores. Esto significa asegurarse de que todos los contenedores que ejecutan diversas cargas de trabajo estén programados para ejecutarse en máquinas físicas o virtuales. Los contenedores deben empaquetarse de manera eficiente y seguir las restricciones del entorno de implementación y la configuración del clúster. Además, *Kubernetes* debe vigilar todos los contenedores en ejecución y reemplazar los contenedores muertos, que no responden o que no están en buen estado.

A un alto nivel, como orquestador de contenedores, *Kubernetes* hace que un grupo de servidores – físicos o virtuales – que ejecutan contenedores, aparezca como un gran servidor lógico que ejecuta contenedores. Como operador, declaramos un estado deseado para el clúster de *Kubernetes* mediante la creación de objetos a través de la API de *Kubernetes*. *Kubernetes* compara continuamente el estado deseado con el estado actual. Si detecta diferencias, toma medidas para garantizar que el estado actual sea el mismo que el estado deseado. Uno de los propósitos principales de un clúster de *Kubernetes* es implementar y ejecutar contenedores, pero también admitir actualizaciones continuas sin tiempo de inactividad mediante técnicas como despliegues *green/blue* y *canary*.

Kubernetes puede programar contenedores, es decir, *Pods* que contienen uno o más contenedores coubicados, en los nodos disponibles en el clúster. Para poder monitorizar el estado de los contenedores en ejecución, *Kubernetes* asume que los contenedores

implementan una sonda de actividad. Si una sonda de actividad informa de un contenedor en mal estado, *Kubernetes* reiniciará el contenedor.

Los contenedores se pueden escalar en el clúster de forma manual o automática mediante un escalador automático horizontal. Para optimizar el uso de los recursos de hardware disponibles en un clúster, por ejemplo, la memoria y la CPU, los contenedores se pueden configurar con cuotas que especifican cuántos recursos necesita un contenedor.

Por otro lado, los límites con respecto a cuánto puede consumir un grupo de contenedores se pueden especificar a nivel de espacio de nombres – *Namespaces* – . Esto es de suma importancia si varios equipos comparten un clúster de *Kubernetes* común.

Otro objetivo principal de *Kubernetes* es proporcionar descubrimiento de servicios de los *Pods* en ejecución y sus contenedores. Los objetos de servicio de *Kubernetes* se pueden definir para el descubrimiento de servicios y también equilibrarán la carga de las solicitudes entrantes sobre los *Pods* disponibles. Los objetos de servicio se pueden exponer de forma externa a un clúster de *Kubernetes*.

Para ayudar a *Kubernetes* a averiguar si un contenedor está listo para aceptar solicitudes entrantes, un contenedor puede implementar una sonda de preparación.

Internamente, un clúster de *Kubernetes* proporciona una gran red IP plana donde cada *Pod* obtiene su propia dirección IP y puede llegar a todos los demás *Pods*, independientemente del nodo en el que se ejecuten. Para admitir varios proveedores de red, *Kubernetes* permite el uso de complementos de red que cumplen con la especificación Container Network Interface – CNI – .

Los *Pods* no están aislados de forma predeterminada, es decir, aceptan todas las solicitudes entrantes. Los complementos CNI que admiten el uso de definiciones de políticas de red se pueden usar para bloquear el acceso a los *Pods*, por ejemplo, solo permitiendo el tráfico de los *Pods* en el mismo espacio de nombres.

Estas son algunas de las capacidades que *Kubernetes* implementa:

- Montaje de sistemas de almacenamiento
- Distribución de secretos/certificados
- Verificación del estado de la aplicación
- Replicación de instancias de aplicación
- Uso de ajuste de escala automático horizontal
- Nombrar y descubrir
- Equilibrio de cargas
- Actualizaciones continuas
- Supervisar recursos
- Acceder y gestionar registros

- Depurar aplicaciones
- Proporcionar autenticación y autorización

3.4.1 Objetos en *Kubernetes*

Kubernetes define una API que se utiliza para administrar diferentes tipos de objetos o recursos, como también se les conoce. Algunos de los tipos o clases más utilizados, son los siguientes [10]:

Node: un nodo representa un servidor, virtual o físico, en el clúster.

Pod: un *Pod* representa el componente implementable más pequeño posible en *Kubernetes*, que consta de uno o más contenedores coubicados. Normalmente, un *Pod* consta de un contenedor, pero existen casos de uso para ampliar la funcionalidad del contenedor principal ejecutando el segundo contenedor en ese mismo *Pod*.

Deployment: el *Deployment* – implementación – se utiliza para implementar y actualizar *Pods*. Los objetos del *Deployment* transfieren la responsabilidad de crear y monitorizar los *Pods* a un *ReplicaSet*. Al crear un *Deployment* por primera vez, el trabajo realizado por el objeto no es mucho más que crear el objeto *ReplicaSet*. El *Deployment* proporciona un mecanismo de actualización que asegura la disponibilidad del recurso, siendo esta totalmente transparente para el cliente, esto lo realiza actualizando los *Pods* de manera gradual y ordenada.

ReplicaSet: se utiliza un *ReplicaSet* para garantizar que se esté ejecutando un número específico de *Pods* en todo momento. Si se elimina un *Pod*, *ReplicaSet* lo reemplazará por uno nuevo.

Service: Un servicio es un *endpoint* de red estable que puede utilizar para conectarse a uno o varios *Pods*. A un servicio se le asigna una dirección IP y un nombre DNS en la red interna del clúster de *Kubernetes*. La dirección IP del servicio permanecerá igual durante la vida útil del servicio. Las solicitudes que se envían a un servicio se reenviarán a uno de los *Pods* disponibles mediante el equilibrio de carga – *Load Balancer* – basado en turnos. De forma predeterminada, un servicio solo se expone dentro del clúster mediante una dirección IP de clúster. También es posible exponer un servicio fuera del clúster, ya sea en un puerto dedicado en cada nodo del clúster o, incluso mejor, a través de un equilibrador de carga externo que conoce *Kubernetes*, es decir, puede aprovisionar automáticamente una dirección IP pública y/o nombre DNS para el servicio. Los proveedores de la nube que ofrecen *Kubernetes* como servicio, en general, admiten este tipo de equilibrador de carga.

Ingress: *Ingress* puede administrar el acceso externo a los servicios en un clúster de *Kubernetes*, generalmente mediante HTTP. Por ejemplo, puede enrutar el tráfico a los servicios subyacentes en función de rutas URL o encabezados HTTP como el nombre de host. En lugar de exponer una serie de servicios de forma externa, ya sea mediante puertos de nodo o mediante equilibradores de carga, en general es más conveniente



configurar un *Ingress* frente a los servicios. Para manejar la comunicación real definida por los objetos *Ingress*, un controlador *Ingress* debe estar ejecutándose en el clúster.

Namespaces: un espacio de nombres se utiliza para agrupar y, en algunos niveles, aislar recursos en un clúster de *Kubernetes*. Los nombres de los recursos deben ser únicos en sus espacios de nombres, pero no entre espacios de nombres.

ConfigMap: *ConfigMap* se usa para almacenar la configuración que usan los contenedores. *ConfigMap* se puede asignar a un contenedor en ejecución como archivo o como variables de entorno.

Secret: se utiliza para almacenar datos confidenciales que utilizan los contenedores, como las credenciales. Los *Secrets* se pueden poner a disposición de los contenedores de la misma manera que *ConfigMaps*. Cualquiera con acceso completo al servidor API puede acceder a los valores de los *Secrets* creados, por lo que no son tan seguros como su nombre podría implicar.

DaemonSet: esto asegura que un *Pod* se esté ejecutando en cada nodo en un conjunto de nodos en el clúster realizando copias del mismo.

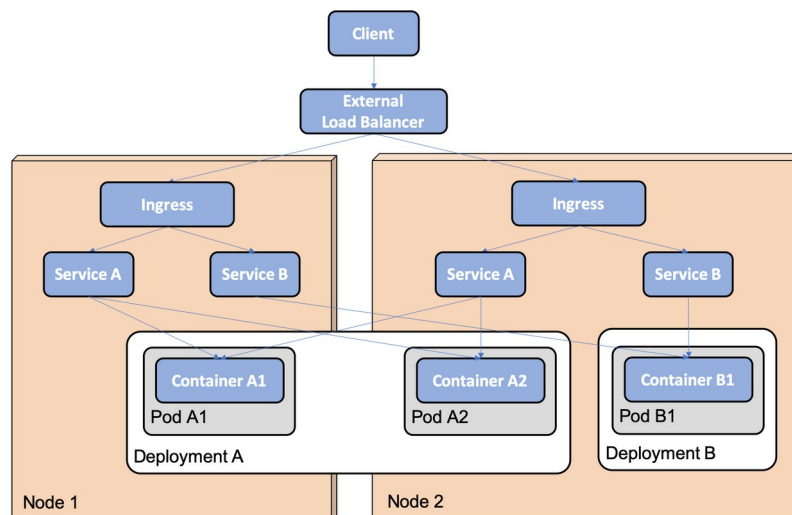


Figura 5. Diagrama de los objetos involucrados en el manejo de solicitudes entrantes.

En el diagrama anterior, podemos ver lo siguiente:

- Se han implementado dos *Deployments*, *Deployment A* y *Deployment B*, en un clúster con dos nodos, Nodo 1 y Nodo 2.
- El *Deployment A* contiene dos *Pods*, *Pod A1* y *Pod A2*.
- El *Deployment B* contiene un *Pod B1*.
- El *Pod A1* está asignado al Nodo 1.
- *Pod A2* y *Pod B1* están asignados al Nodo 2.
- Cada *Deployment* tiene un servicio correspondiente implementado, *Service A* y *Service B*, y están disponibles en todos los nodos.
- Un *Ingress* se define para enrutar las solicitudes entrantes a los dos servicios.

3.4.2 Archivos de creación de objetos

Kubernetes usa archivos YAML como entradas para la creación de objetos como *Pods*, *ReplicaSets*, *Deployments* o *Services*. Estos archivos deben estar bien formados e indentados. Los archivos contienen la información necesaria para crear el objeto requerido, aquí definimos la imagen *Docker* de nuestro ARS a cargar en un *Deployment* (fig. 6), o en el caso de los *Services* que puerto queremos exponer (fig. 7).

```

1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: erp-dpl
5   namespace: embalpack
6   labels:
7     app: erp
8     type: DevOps
9 spec:
10  template:
11    metadata:
12      name: erp-pod
13      annotations:
14        prometheus.io/scrape: "true"
15        prometheus.io/port: "8080"
16        prometheus.io/path: "actuator/prometheus"
17      labels:
18        app: erp
19        type: DevOps
20    spec:
21      containers:
22        - name: erp-container
23          image: tambori.dsic.upv.es:10050/embalpack/erp:1.0
24          ports:
25            - name: port8080
26              containerPort: 8080
27            - name: port8081
28              containerPort: 8081
29            - name: port8082
30              containerPort: 8082
31          imagePullSecrets:
32            - name: regcred
33      replicas: 1
34      selector:
35        matchLabels:
36          type: DevOps
37
38

```

Figura 6. Archivo de creación de un *Deployment*

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: pl1-ars-svc
5   annotations:
6     prometheus.io/scrape: "true"
7     prometheus.io/port: "8080"
8     prometheus.io/path: "actuator/prometheus"
9 spec:
10  type: NodePort
11  ports:
12    - targetPort: 8080
13      port: 30020
14      nodePort: 30020
15  selector:
16    app: pl1
17    type: DevOps

```

Figura 7. Archivo de creación de un *Service nodePort*.

3.4.3 Kubernetes CLI, *kubectl*

kubectl es la herramienta CLI – *Command Line Interface* – de *Kubernetes*. Una vez que se ha configurado un clúster, esta suele ser la única herramienta que se necesita para administrar el clúster. *kubectl*, permite ejecutar comandos en clústeres de *Kubernetes*. Se puede usar *kubectl* para desplegar aplicaciones, inspeccionar y administrar recursos del clúster y ver registros. Esto lo consigue con llamadas al *api-server* de *Kubernetes* de una forma totalmente transparente para el usuario.

Para la configuración, *kubectl* busca un archivo llamado ‘config’ en el directorio `$HOME/.kube`. En este archivo entre otros datos se encuentran los certificados de creación del clúster, los cuales utiliza *kubectl* para acceder al *api-server* de *Kubernetes*.

Algunos usos de *kubectl*:

- Despliegue → `kubectl create -f fileDeploy.yaml`
- Borrado → `kubectl delete pod erp-pod`
- Inspeccion → `kubectl describe pod erp-pod`
- Listado → `kubectl get pods,deployments`

3.5 Embalpack

En los últimos 4-5 años el grupo Tatami del Centro PROS de la UPV mantiene una colaboración con la empresa Embalpack, en la cual asesoran en el diseño de una solución IoT/Industria 4.0. Entre estos trabajos, el grupo Tatami tiene uno más centrado en dotar de capacidades de autogestión a su infraestructura computacional. La estrategia se centra en convertir ciertos servicios en auto-adaptables, respondiendo estos automáticamente a eventos definidos.

Embalpack Levante S.L. es una fábrica situada en Favara (Valencia), que se dedica a la producción de cantoneras de cartón a partir de un proceso de conformado de papel.

En Embalpack se dispone de la siguiente infraestructura. Esta está capacitada para comunicarse como si se tratara de un dispositivo IoT.

Recursos:

- Bobinas de papel: Con capacidad de comunicación con el entorno. Hay servicios que permiten almacenar su información, otros atienden a su localización.
- Operarios: También generan información de su participación en la máquina o en la operación que están realizando.

M. Blanco Máñez

- Carro Elevador: Empleados para llevar las bobinas de papel de una ubicación a otra.
- Equipos: Tenemos seis equipos encargados de gestionar las cadenas de producción (cuatro para las máquinas cantoneras y dos para las máquinas rebobinadoras) y un equipo para el ERP.
- Controladores: Cuatro controladoras/equipos de bajo rendimiento, encargados de gestionar los semáforos de las cadenas de producción.

Máquinas:

- Máquinas rebobinadoras: rebobina bobinas de papel en bobinas más pequeñas. Atienden órdenes de producción – que vienen desde el ERP de la empresa –, y reportan en tiempo real – plataforma IOT – su estado.
- Máquinas que hacen cantoneras: Producen cantoneras a partir de bobinas generadas por las rebobinadoras. También generan información sobre la producción, atendiendo a las órdenes de producción generadas por el ERP.

(Micro) Servicios:

- Gestión de stock: Controla el stock de bobinas y cantoneras.
- Gestión de recursos: Controla los recursos disponibles.
- Gestión de localizaciones: Controla la ubicación de los recursos.
- ERP de producción: Este no es un microservicio, pero se ha desarrollado infraestructura para que se pueda conectar a la plataforma IoT a través de una API REST.

Además, como se pretende trabajar siguiendo la estética de los dispositivos IoT, donde cada dispositivo cuenta con su versión digital donde almacena su estado actual, se ha decidido incluir una serie de microservicios extra para controlar los diversos aspectos de los recursos y así aumentar la funcionalidad del sistema y acercarlo más a lo que sería una típica infraestructura de IoT:

- Gestión del registro: Registra todos los objetos digitales de los que disponemos.
- Objetos digitales: Almacena el estado del objeto digitalizado.
- Registro de las transformaciones: Registra las transformaciones que sufren las bobinas, producidas por las máquinas.



4. Análisis del problema

Para ejemplificar la propuesta se utiliza un caso de estudio real en el que se aplican los principios teóricos (ver 3) para desarrollar el sistema informático – que utilizará componentes *adaptive-ready* – de una fábrica de producción industrial. Se tienen pues que diseñar e implementar un conjunto de servicios *adaptive-ready* y se ha de desplegar una Arquitectura de microservicios reconfigurable sobre *Kubernetes*.

Para esto se requiere de un análisis del problema y de la definición de los requisitos de la solución. Esto nos servirá de guía en la posterior fase de diseño eligiendo la función que debe realizar el software y establecer o indicar cuál es la forma más adecuada para la implementación.

En el contexto del problema tenemos por un lado los microservicios que deberán exponer una interfaz de adaptación, por otro el orquestador que será la infraestructura donde desplegaremos nuestra arquitectura, un registro de contenedores, el *Efactor* que actuará directamente sobre *Kubernetes* y los microservicios, y por último la *Sonda* que reportará al monitor de MAPE-K el estado de los dispositivos virtuales que forman parte de la producción de la fábrica y que afectan directamente a la configuración de la arquitectura de los microservicios. En este caso en particular serán semáforos que indican el ritmo de producción. Tanto el *Efactor* como la *Sonda* serán una extensión del bucle de control (fig. 9).

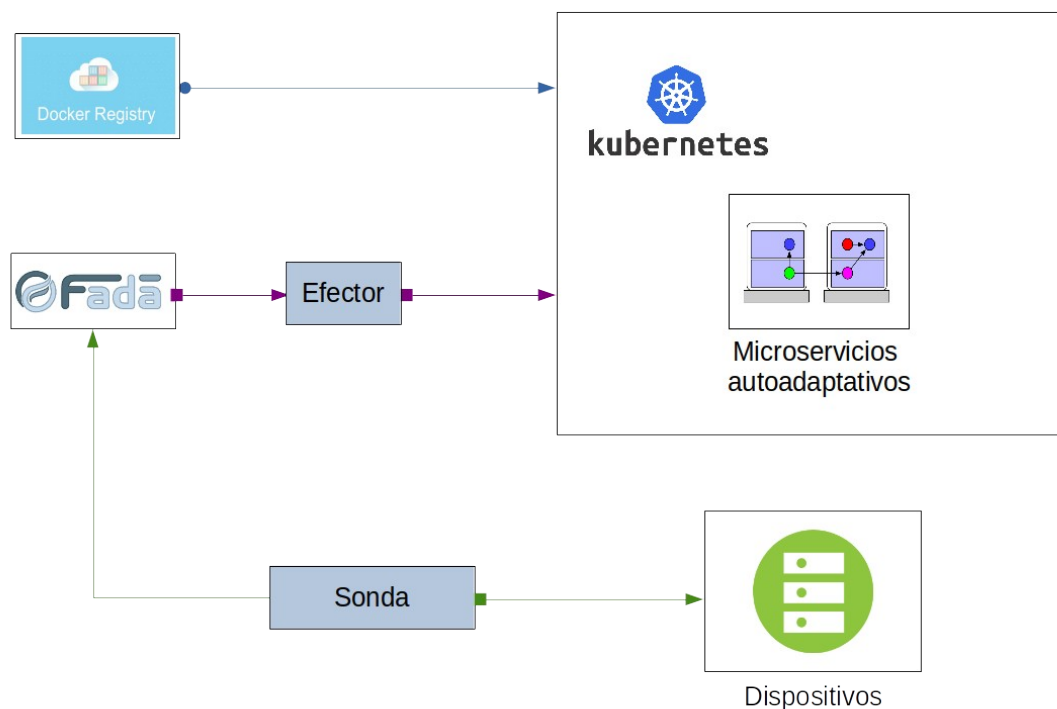


Figura 9. Esquema básico del contexto del problema.

4.1 Servicios *adaptive-ready* ARS/ARM.

Ahora que tenemos definido el esquema básico, en primer lugar analizaremos los microservicios. Como hemos visto en 3.3 para ser auto-adaptables los microservicios que ofrecen la funcionalidad de la aplicación deben estar encapsulados dentro de un ARM . Este ARM debe ofrecer una interfaz de comunicación que podrá crear instancias de los microservicios y exponer u ocultar interfaces de los mismos. Así mismo debe poder enlazar unos microservicios con otros si así lo requiere la configuración (*fig. 10*).

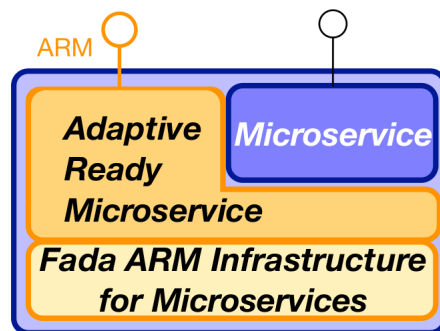


Figura 10. Partes básicas de un ARM en la que expone la interfaz que habilitará la auto-adaptación.

4.1.1 Requisitos. Contrato de adaptación.

Para convertir los microservicios en *adaptive-ready*, se va a definir un controlador de adaptación, para ello introducimos el concepto ‘contrato de adaptación’. El contrato de adaptación [11] establece el mecanismo a través del cual los bucles de control auto-gestionarán las arquitecturas de microservicios en base a unas operaciones de adaptación que realizarán la reconfiguración arquitectónica en tiempo de ejecución (*fig. 11*).

Estas operaciones del controlador de adaptación de los microservicios serán:

- Desplegar M : se crea (instancia) del microservicio M
- Eliminar M : se elimina el microservicio M
- Exponer I de M : se pone accesible la interfaz I del microservicio M
- Ocultar I de M : se elimina la accesibilidad a la interfaz I del microservicio M
- Enlazar I(r) de M1 con I(p) de M2 : se conecta la interfaz requerida I(r) del microservicio M1 con la interfaz expuesta I(p) del microservicio M2
- Desenlazar I(r) de M1 de I(p) de M2 : se elimina el enlace entre la interfaz requerida I(r) de M1 de la interfaz expuesta I(p) de M2

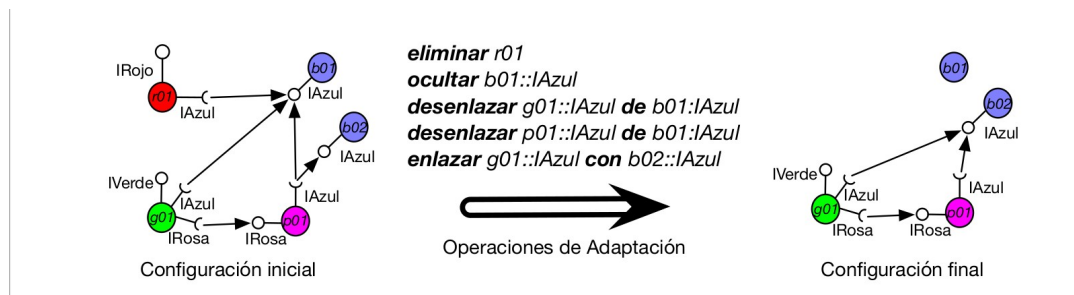


Figura 11. Ejemplo de cambio arquitectónico de una configuración inicial a una final, a través de las operaciones de adaptación [11].

Definiremos un ARM como un microservicio que se extiende para implementar el contrato de adaptación. Este contrato se materializa a través de una interfaz de adaptación que expone las operaciones de adaptación. Esta interfaz de adaptación será utilizada por el bucle de control para reconfigurar la arquitectura de microservicios.

Para cumplir con los requisitos y crear un ARM, deberemos pues implementar 6 métodos básicos que expondrán estas operaciones a través de una API REST.

Como requisito añadido este ARM deberá crear en su arranque una instancia de un microservicio, posteriormente esta instancia será configurada – adaptada – por el propio ARM.

4.2 Kubernetes

Kubernetes proporcionará una plataforma sobre la que desplegar la arquitectura. Aprovecharemos toda la funcionalidad que contiene para la implementación de nuestro escenario.

4.2.1 Requisitos

Se van a utilizar las mismas operaciones descritas en 4.1.1 para la adaptación de los microservicios, pero se interpretarán como adaptación de la arquitectura.

Así pues se deberán generar los archivos de creación para:

- Cuando se requiera Desplegar:

- Se creará un *Deployment* (ver 3.4.1) del ARM, se establecerá una réplica en *ReplicaSet* (ver 3.4.1) para el ARM.
- Se creará un *Service* (ver 3.4.1) que expondrá la interfaz de adaptación.

- Cuando se requiera Replegar :

- Se borrará el *Deployment* requerido.
- Se borrará el *Service* asociado a ese *Deployment*.

- Cuando se requiera Exponer :

- Se creará un *Service* que expondrá la interfaz de la instancia del microservicio creado.



- Cuando se requiera Ocultar :
 - Se borrará el *Service* que expone la interfaz de la instancia del microservicio creado.

4.4 Efactor

Este componente será el responsable de ejecutar las órdenes del bucle MAPE-K sobre *Kubernetes* y los servicios *adaptive-ready*. Cuando las órdenes se refieran a cambios en la arquitectura actuará sobre *Kubernetes*, cuando estas se refieran a la configuración de los microservicios – exponer, enlazar, ocultar, desenlazar – actuará sobre los servicios *adaptive-ready*.

4.4.1 Requisitos

El *Efactor* deberemos diseñarlo e implementarlo tal que:

- Utilice *api-server* de *Kubernetes* para comunicarse con el orquestador.
- Utilice la API REST definida en nuestros ARM para comunicarse con estos.
- Cuando se requiera Desplegar:
 - Deberá buscar el archivo YAML de creación (*ver 3.4.2*) del *Deployment* que coincida con el nombre del ARM. A continuación creará el *Deployment* correspondiente.
 - Deberá buscar el archivo YAML de creación de *Service* que coincida con el nombre del servicio ARM. A continuación creará el *Service* correspondiente.
- Cuando se requiera Replegar :
 - Se borrará el *Deployment* requerido.
 - Se borrará el *Service* requerido.
- Cuando se requiera Exponer :
 - Deberá buscar el archivo YAML de creación de *Service*. A continuación creará el *Service* correspondiente. Posteriormente realizará la operación Exponer sobre el microservicio .
- Cuando se requiera Ocultar :
 - Se borrará el *Service* que expone la interfaz de la instancia del microservicio creado. Posteriormente realizará la operación Ocultar sobre el microservicio .
- Cuando se requiera Enlazar :
 - Realizará la operación Enlazar sobre el microservicio.
- Cuando se requiera Desenlazar :
 - Realizará la operación Desenlazar sobre el microservicio.

4.5 Sonda

Esta *Sonda* deberá reportar el estado de los dispositivos virtuales en nuestro escenario, los datos los enviará al bucle MAPE-K y más concretamente al monitor (ver 3.1.1.1). Básicamente controlara si estos dispositivos están operativos.

4.5.2 Requisitos

Esta *Sonda* deberá pues:

- Controlar el estado de los dispositivos virtuales que se requieran.
- Será de función continua. Es decir empezará a tomar mediciones una vez se determine el dispositivo a medir, independientemente si este es accesible o no.
- Utilizará mensajes del tipo `status=Healthy`, que irá enviando periódicamente al monitor.

4.6 Registro de contenedores

El registro de contenedores será un repositorio privado de imágenes *Docker* de los ARM creados. Este repositorio lo utilizará *Kubernetes* para cargar las imágenes y crear así los *Deployments*.

El único requisito que se demanda es que sea de acceso privado, esto conlleva realizar configuraciones de seguridad basadas en certificados. Se deberán generar certificados y auto-firmarlos ya que para la realización de este trabajo no disponemos de un certificado avalado por una autoridad certificadora – CA –. Esto es una práctica habitual en la fase de desarrollo.



5. Diseño de la solución

Iniciaremos el diseño con el *adaptive-ready microservice* ARM, estableciendo las operaciones que debe realizar y que funcionalidad va a proporcionar, definiremos el API REST que se utilizará en la comunicación y crearemos los mensajes indicando su formato y contenido.

Para continuar nos centraremos en el *Efactor*, donde expondremos las funciones que va a realizar sobre *Kubernetes* y los *adaptive-ready microservices*. Al igual que los *adaptive-ready* se van a diseñar como microservicio, por lo tanto determinaremos el API REST que utilizará *FADA* para enviar sus órdenes. También estableceremos el formato de los mensajes que se van a tratar.

Para finalizar diseñaremos la *Sonda*, en la que determinaremos cómo va a funcionar, las operaciones que va a realizar y cómo se activará cada *Sonda*. Comentaremos de qué forma se va a comunicar con el monitor en *FADA*.

5.1 Servicio *adaptive-ready*

En este apartado vamos a realizar el diseño de los ARM, *Adaptive-Ready Microservice*.

5.1.1 Operaciones *adaptive-ready*

Para cumplir con el contrato de adaptación descrito en 4.1.1, el diseño de las operaciones de adaptación debe seguir una serie de convenciones.

- La operación de desplegar un microservicio es la responsable de configurar y arrancar el microservicio.
- Al iniciar un microservicio ARM, se deberá lanzar la ejecución de la operación desplegar, y luego publicar (de manera accesible) la interfaz de adaptación.
- La operación de eliminar un microservicio realiza la operación inversa. Las operaciones de enlazar y desenlazar, deben realizar cambios sobre las dependencias del microservicio con otros servicios.
- Las operaciones de exponer y ocultar interfaces, deben activar y parar, respectivamente, las implementaciones de dichas interfaces proporcionadas.



Para ello el ARM presentará estas funciones:

Función	Operación a realizar
deploy	Crea una instancia del microservicio → Desplegar.
undeploy	Borra la instancia requerida del microservicio → Eliminar.
expose	Pone accesible la interfaz de la instancia → Exponer.
hide	Elimina la accesibilidad a la interfaz de la instancia → Ocultar.
bind	Conecta la interfaz requerida con la interfaz expuesta de otro microservicio → Enlazar.
unbind	Desconecta la interfaz requerida con la interfaz expuesta de otro microservicio → Desenlazar.

En esta figura (fig. 12) se muestra como es la secuencia de las funciones deploy, expose y bind.

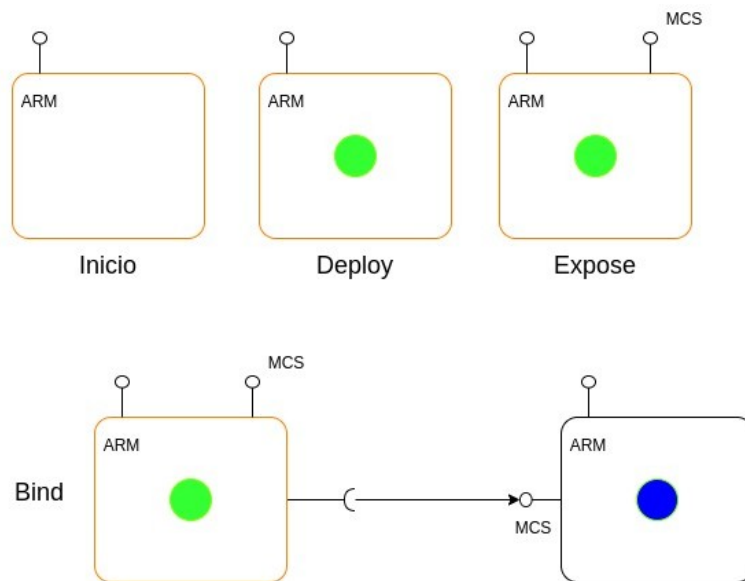


Figura 12. Esquema del resultado de algunas de las funciones que realiza ARM.

La función deploy se realizará al inicio de la ejecución de ARM, creando así una instancia ‘vacía’ de un microservicio , posteriormente este microservicio será adaptado por ARM.

5.1.2 Definición de API REST

Una API, o interfaz de programación de aplicaciones, es un conjunto de reglas que definen cómo las aplicaciones o los dispositivos pueden conectarse y comunicarse entre sí. Una API REST es una API que se ajusta a los principios de diseño de REST [12], o

estilo arquitectónico de transferencia de estado. Por este motivo, las API REST a veces se denominan API RESTful.

Por lo tanto para que las funciones de ARM sean accesibles desde aplicaciones externas y dar así acceso a los recursos, vamos a definir unas reglas que deberán respetar tanto el ARM como la contra-parte cliente. Esta API REST la utilizará el *Efector* para comunicarse con ARM.

Para desarrollar una API REST es necesario establecer un contrato formal – *Uniform contract* – el cual se basa en tres elementos principales [13].

- Sintaxis del identificador de recursos: Expresa hacia dónde se transfieren los datos y desde dónde provienen. Representados por URI's / URL's.
- Métodos: Mecanismos de protocolo utilizados para transferir los datos – GET, PUT, DELETE , POST, OPTIONS ... – .
- Media type: Formato de los datos que se están transfiriendo – text/plain, application/XML, application/JSON ... – .

Sintaxis y métodos : En la siguiente tabla se refleja la elección de las URL's y los métodos HTTP a utilizar. El recurso raíz se encontrará en `http://<IP del servidor>/` .

URL	Método	Función
<code>/ARM/{serviceId}/services/{type}</code>	GET	Muestra toda la información del microservicio.
<code>/ARM/{serviceId}</code>	PUT	Da acceso a las funciones de ARM con esa identificación.

Un mensaje GET válido sería:

`http://192.168.99.111:30092/ARM/PL1/services/PL`

Un mensaje PUT válido sería:

`http://192.168.99.111:30092/ARM/PL1`

Media type : Definición del formato de los datos que se van a manejar.

Todos los mensajes tendrán formato JSON tanto los de respuesta como los entrantes.

Plantilla del mensaje de entrada – PUT – :

```
{ "action": "deploy" / "expose" / "unbind" / .....,
  "require": [ { "id": "TLC", "url": "http://tlc1-service.svc:8080/TLC/TLC1" } ],
  "expose": [ { "Interface": "hello", "Available": true }, { "Interface": "ERP", "Available": true } ] }
```

Donde:

"action" → determina la función que el ARM debe ejecutar.

"require" → determina la URL donde se debe enlazar el microservicio. En el caso de unbind este espacio se deja en blanco.

"expose" → determina la interfaz o interfaces que el microservicio ha de exponer. En el caso de hide este espacio se deja en blanco.

5.2 Efactor

Como hemos visto en 4.4.1 este componente se va a comunicar con *Kubernetes* para la implementación de la arquitectura que se requiera en ese momento y con el bucle MAPE-K que será el que le envíe las funciones a realizar sobre ARM.

Se va a diseñar como un microservicio ofreciendo así todas las características de este tipo de componentes.

5.2.1 Funciones del Efactor

Las funciones del *Efactor* serán análogas a las funciones de ARM (ver 5.1.1) por lo que quedarán de la siguiente manera:

Función	Operación a realizar
deploy	Crea un <i>Deployment</i> y un <i>Service</i> en <i>Kubernetes</i> . Se habrá creado automáticamente la instancia del microservicio
undeploy	Borra <i>Deployment</i> y <i>Service</i> en <i>Kubernetes</i> . Se habrá borrado automáticamente la instancia del microservicio
expose	1. Creará el <i>Service</i> correspondiente en <i>Kubernetes</i> . 2. Realizará la operación expose sobre ARM.
hide	1. Borrará el <i>Service</i> correspondiente en <i>Kubernetes</i> . 2. Realizará la operación hide sobre ARM.
bind	Realizará la operación bind sobre ARM.
unbind	Realizará la operación unbind sobre ARM.

5.2.2 Definición de API REST

A la hora de definir el API REST para comunicarse con el componente se va a utilizar por cuestiones de eficiencia el mismo API definido para ARM (ver 5.1.2)

Por que lo sería de la siguiente forma:

Sintaxis y métodos : En la siguiente tabla se refleja la elección de las URL's y los métodos HTTP a utilizar. El recurso raíz se encontrará en `http://<IP del servidor>/` .

URL	Método	Función
<code>/ARM/{serviceId}</code>	PUT	Actúa sobre <i>Kubernetes</i> . Da acceso a las funciones de ARM.

Un mensaje PUT válido sería:

```
http://localhost:10060/ARM/PL1
```

Media type : Definición del formato de los datos que se van a manejar.

Todos los mensajes tendrán formato JSON, que contendrán los datos que utilizará el *Efactor*.

Plantilla del mensaje para `deploy/undeploy`:

```
{ "type": "ERP", "action": "deploy" }
```

Donde:

"type" → determina el tipo de ARM sobre que se tiene que actuar.

"action" → determina la función que el *Efactor* debe ejecutar.

Plantilla del mensaje para `expose/hide`:

```
{ "type": "ERP", "action": "expose", "interface": "PLWO" }
```

Donde:

"type" → determina el tipo de ARM sobre que se tiene que actuar.

"action" → determina la función que el *Efactor* debe ejecutar.

"interface" → determina la interfaz o interfaces que el microservicio ha de exponer.

Caso "action": "hide" → determina la interfaz o interfaces que el microservicio ha de ocultar.



Plantilla del mensaje para `bind/unbind`:

```
{"type":"ERP", "action":"bind", "supply":"http://tlc1-service.svc:8080/TLC/TLC1"}
```

Donde:

"type" → determina el tipo de ARM sobre que se tiene que actuar.

"action" → determina la función que el *Efector* debe ejecutar.

"supply" → determina la URL al que el microservicio se ha de enlazar/desenlazar.

5.2.3 Traducción de mensajes

Como ya se ha comentado este componente será el punto de unión entre *Kubernetes*, *FADA* y los ARM.

El método que utiliza *FADA* para enviarle mensajes al *Efector* ha sido desarrollado independientemente a este trabajo, si bien se pactó el formato, en los contenidos de los mensajes existen mínimas diferencias – exclusivamente en las etiquetas –, que deberemos procesar. Esto ocurre habitualmente en la fase de desarrollo de un proyecto, o en la integración de aplicaciones cuando incluso se utilizan distintos protocolos por las partes.

5.3 Sonda

Este componente debe medir el estado actual de los dispositivos y reportar estas mediciones periódicamente al monitor de MAPE-K.

Se va a diseñar como un microservicio ofreciendo así todas las características de este tipo de componentes..

5.3.1 Funciones de la Sonda

Este componente constará de una sola función:

Función	Operación a realizar
startProbe	Mide el estado actual del dispositivo objetivo

5.3.2 Definición de API REST

Como este componente se va a implementar como un microservicio definiremos la API para comunicarnos con el microservicio desde un aplicación externa.

Sintaxis y métodos : En la siguiente tabla se refleja la elección de las URL's y los métodos HTTP a utilizar. El recurso raíz se encontrará en `http://<IP del servidor>/` .

URL	Método	Función
/start	PUT	Inicia una <i>Sonda</i> sobre el objetivo contenido en el mensaje

Un mensaje PUT válido sería:

```
http://localhost:10090/start
```

Media type : Definición del formato de los datos que se van a manejar.

Todos los mensajes tendrán formato JSON, que contendrán los datos que utilizará la *Sonda*.

Plantilla del mensaje para `startProbe`:

```
{ "target":{ "id": "DevPL", "url": "http://localhost:10061/actuador/health" } }
```

Donde:

"id" → determina el tipo de ARM sobre que se tiene que actuar.

"url" → determina la URL objetivo donde se medirá el estado actual.

Si observamos "url" veremos que es muy descriptiva – `/actuador/health` –, esto es porque vamos a utilizar una de las funcionalidades que ofrece *Spring* a la hora de desarrollar microservicios, se trata de la dependencia `actuator` .

Esta dependencia habilita un puerto y una URL para que de manera transparente podamos medir el estado del microservicio.

La respuesta está en formato JSON y contiene lo siguiente:

```
{ "status": "UP" }
```

o

```
{ "status": "DOWN" }
```

Evidentemente cuando el microservicio está caído no hay respuesta, la *Sonda* tratará entonces la excepción generada y reportará la caída del microservicio.



5.3.3 Comunicación con FADA

La *Sonda* deberá reportar las mediciones periódicamente al monitor MAPE-K que opera en *FADA*. Es por ello que hay que definir de que forma lo hacemos y qué van a contener los mensajes.

FADA va a recibir los datos de la *Sonda* utilizando colas de mensajería, esto comporta la creación de un publicador en el componente, *FADA* entonces se suscribirá a ese topic.

El *broker* de comunicación será `tcp://tambori.dsic.upv.es:10083` en el topic `"embalpack/monitors/health"`.

5.3.3.1 Definición de mensajes

El formato acordado para los mensajes es JSON y contendrá lo siguiente:

```
{"health-check":"H","microservice":"DevTL1"}
```

Donde:

"health-check" → determina los valores correspondientes al estado que son H, U y D (Healthy, Unhealthy y Dead).

Estos valores tendrán siguiente correspondencia con los valores recibidos:

Estado	Valor		
Recibido	UP	DOWN	CONN . REFUSED
Enviado	H	U	D

"microservice" → determina el microservicio del que se está reportando información.

5.4 Resumen del diseño

Para ver como funcionará nuestro diseño haremos una traza de un despliegue inicial de un conjunto de servicios *adaptive-ready* sobre una arquitectura de microservicios reconfigurable sobre *Kubernetes*. Usaremos diagramas para ilustrar el texto explicativo.

Lo haremos en cuatro partes: Inicio, despliegue en *Kubernetes*, operaciones sobre ARM y operaciones auto-adaptativas.



5.4.1 Inicio

Iniciamos la ejecución poniendo en funcionamiento *Minikube*, *FADA*, *Efactor*, los dispositivos y la *Sonda* (fig. 13).

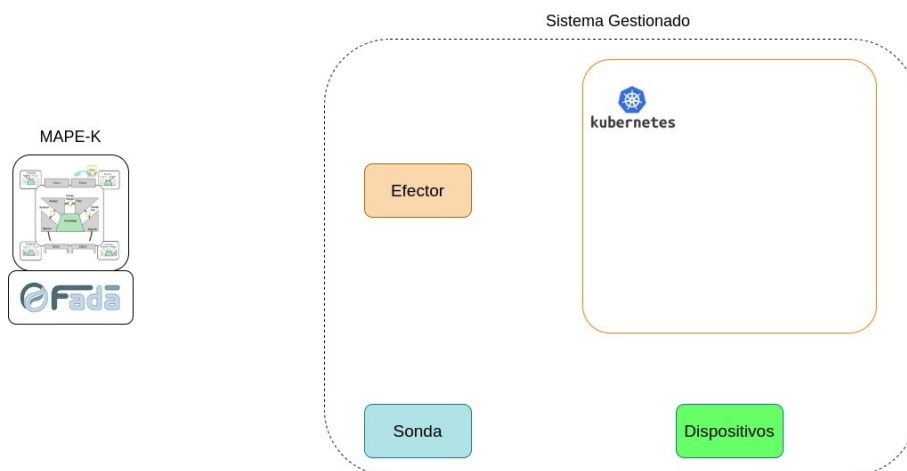


Figura 13. Escenario inicial. Arranque de componentes.

5.4.2 Despliegue en Kubernetes

A continuación *FADA* envía la orden *deploy* a *Efactor*, y este desplegará en *Kubernetes* el ARM correspondiente, creando a su vez el *Service* asociado. Recordemos que el ARM en su arranque crea una instancia básica del microservicio. En este momento la *Sonda* ya está enviando datos a *FADA* (fig. 14).

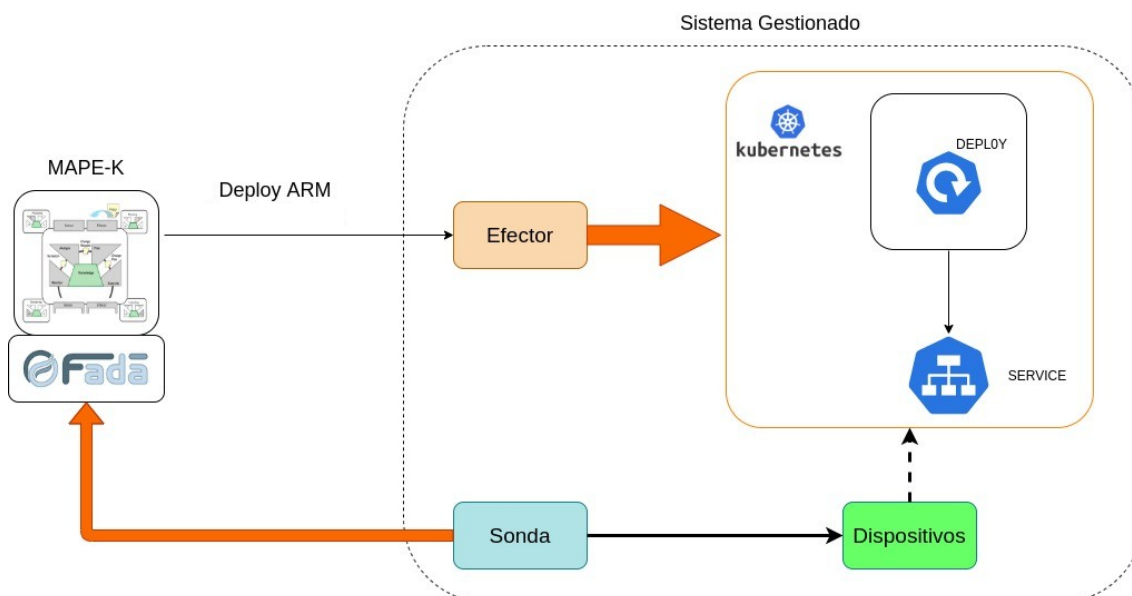


Figura 14. Deploy en Kubernetes. MAPE-K envía la orden 'deploy' al Efactor. Este actúa sobre Kubernetes realizando las operaciones oportunas.



5.4.3 Operaciones sobre ARS/ARM

Ahora *FADA* comenzará a configurar – adaptar – el microservicio, determinando las interfaces que exponer o los enlaces a realizar. Para ello enviará estas órdenes de configuración a *Efactor* el cual actuará sobre ARM en consecuencia (fig. 15).

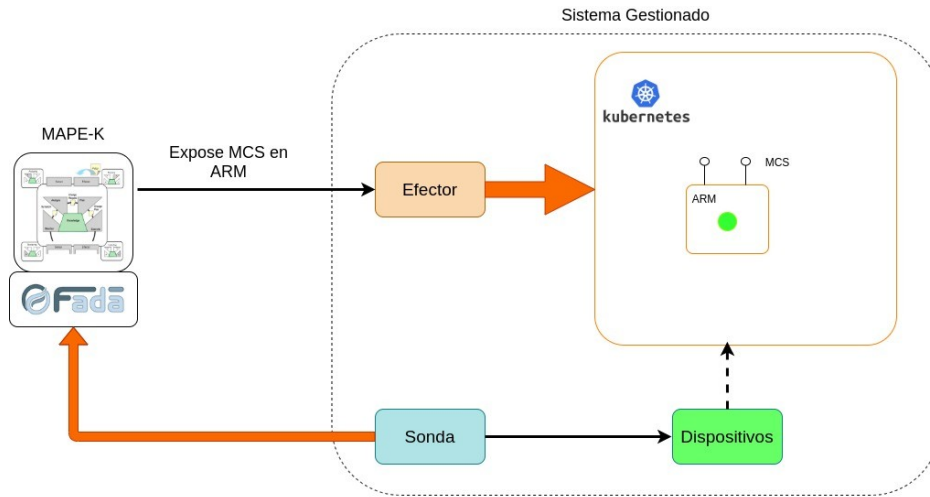


Figura 15. Operaciones sobre ARM. MAPE-K envía la orden ‘expose’. El Efactor creará un Service en Kubernetes y expondrá la interfaz del microservicio.

5.4.4 Operaciones auto-adaptativas

Así pues una vez este desplegado todo el sistema podremos, siguiendo los mismos pasos, adecuar tanto la arquitectura actuando sobre *Kubernetes* como la configuración de los microservicios .

De esta manera nuestro sistema reaccionará ante cualquier evento que el bucle MAPE-K determine que es susceptible de tener en cuenta. En caso de caída de un dispositivo virtual, la *Sonda* lo detectará y reportará al monitor. Este evento se analizará para después realizar un plan de adaptación. Se crearán unas reglas que el ejecutor transformará en órdenes que enviará a *Efactor* para ejecutarlas (fig. 16).

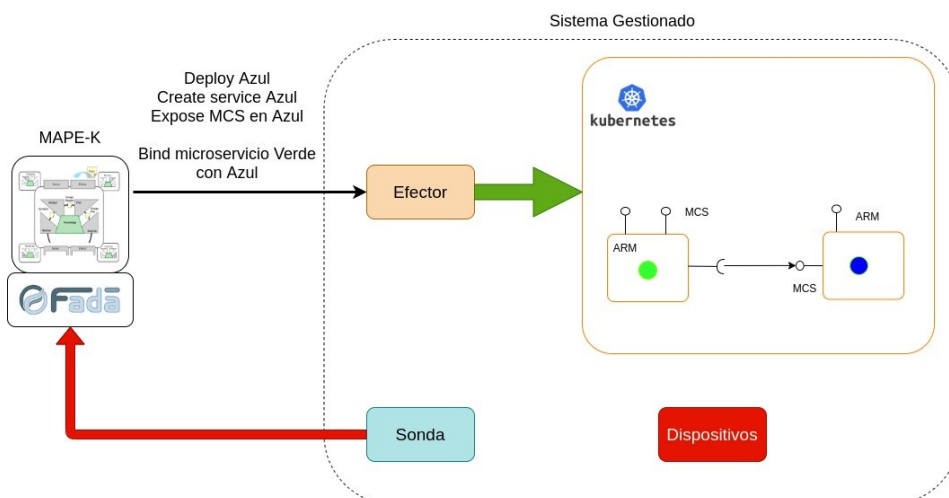


Figura 16. Operaciones auto-adaptativas. Cuando el dispositivo se apaga, MAPE-K envía órdenes al Efactor para realizar la adaptación.

6. Implementación

La implementación se divide en tres fases, una primera donde describiremos los pasos seguidos para realizar un despliegue inicial en *Minikube*, la segunda donde implementaremos el *Efector* y la *Sonda* y una última donde explicaremos la implementación final donde entra en juego el bucle MAPE-K.

6.1 Despliegue inicial en *Minikube*

Para realizar esta implementación necesitamos un ARM, su imagen *Docker*, alojarla en *Docker registry* y los archivos de creación de objetos de *Kubernetes*.

6.1.1 Implementación de ARS/ARM

Vamos a utilizar *Spring Boot* (ver 2.1) para implementar el *adaptive-ready microservice*. Empezaremos creando el proyecto con *Gradle* incluyendo las dependencias que creemos serán adecuadas (fig. 17).

```
dependencies {  
    implementation 'org.json:json:20201115'  
    implementation 'org.springframework.boot:spring-boot-starter-actuator'  
    implementation 'org.springframework.boot:spring-boot-starter-validation'  
    implementation 'org.springframework.boot:spring-boot-starter-web'  
    runtimeOnly 'io.micrometer:micrometer-registry-prometheus'  
    testImplementation 'org.springframework.boot:spring-boot-starter-test'  
}
```

Figura 17. Dependencias Gradle en el proyecto creado en Spring.

De entre estas dependencias hay que destacar *spring-boot-starter-web* que nos proporciona el framework para la realización de servicios REST y *spring-boot-starter-actuator* que como se ha dicho en 5.3.2 nos habilita un puerto y una URL donde medir el estado del microservicio.

Utilizaremos esta estructura de paquetes y clases (*fig. 18*):

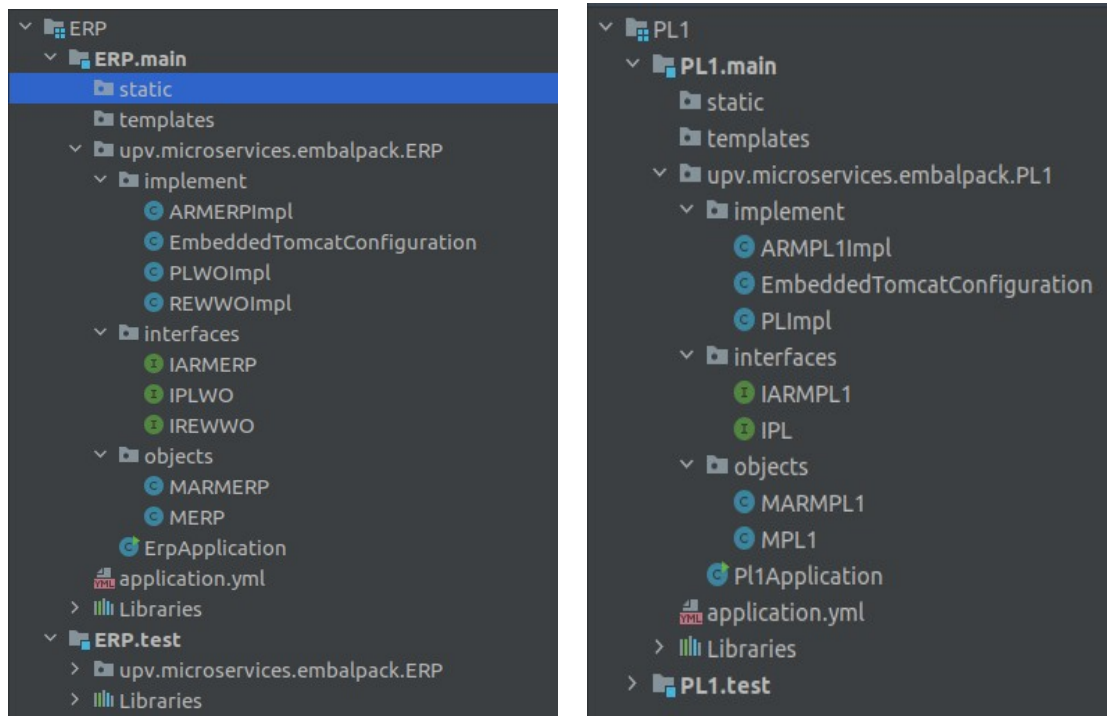


Figura 18. Estructura de paquetes y clases de dos adaptive-ready microservice ARM.

Donde:

- El paquete `interfaces` contiene las clases que reciben los datos, en estas se declara la URL que se tiene que utilizar para ARM (*fig. 19*) o para los microservicios. Servirán como *endpoint* – interfaz expuesta – tanto para el ARM como para los microservicios, recordemos que al inicio sólo se expondrá la de ARM.

```
package upv.microservices.embalpack.ERP.interfaces;

import ...

public interface IARMERP {

    @GetMapping(value = "/ARM/{microserviceId}/services/{type}")
    public ResponseEntity<String> getService(@PathVariable(value = "microserviceId") String id, @PathVariable(value = "type") String type);
    @PutMapping(value = "/ARM/{id}")
    public ResponseEntity<String> armActionApply(@PathVariable(value = "id") String instance, @RequestBody String objjson);
}
```

Figura 19. Declaración de la interfaz y las URL's para API REST de ARM.

- El paquete `implement` contiene las clases que implementan las funciones de esas `interfaces`, tanto para ARM como para los microservicios. En el caso de ARM aquí se procesarán los mensajes JSON que contienen la información, p.e deploy de ERP, o expose la interfaz I de ERP (fig. 20).

```
@Override
public ResponseEntity<String> armActionApply( String instance,String objson) {
    MARMERP arm= ErpApplication.arm;
    ResponseEntity<String> respuesta;
    JSONObject requestValue = new JSONObject(objson);
    String action = requestValue.getString( key: "action");
    JSONArray jsonArrayRequire;
    JSONArray jsonArrayExpose;
    JSONArray jsonArraySetParams;

    switch (action) {
        case "deploy":

            jsonArrayRequire = new JSONArray(requestValue.getJSONArray( key: "require"));
            jsonArrayExpose = new JSONArray(requestValue.getJSONArray( key: "expose"));

            if(arm.deploy(instance, jsonArrayRequire, jsonArrayExpose)) {
                respuesta = ResponseEntity.status(HttpStatus.CREATED).body("Service Created\n ID:");
            } else {
                respuesta = ResponseEntity.badRequest().build();
            }
            System.out.println("Deploy action done");
            break;

        case "bind":
            jsonArrayRequire = new JSONArray(requestValue.getJSONArray( key: "require"));
    }
```

Figura 20. Fracción de código de la implementación de la función declarada en la interfaz.

La clase `EmbeddedTomcatConfiguration` nos permite abrir más de un puerto en el microservicio, algo necesario si se quiere separar los `endpoints` de ARM de los del microservicio.

- El paquete `object` contiene las clases que definen los objetos que se van a crear. Durante la ejecución se crea una única instancia de ARM, no así de los microservicios que, a parte de la instancia inicial que se crea a la vez que ARM, se crearán tantas instancias como requiera la producción en ese momento. En el objeto MARM es donde residen las funciones de ARM vistas en 5.1 (fig. 21).

```
package upv.microservices.embalpack.PL1.objects;

import ...

public class MARMPL1 {
    private final Map<String, MPL1> services = new HashMap<>();
    private final Map<MPL1, Map<String, Boolean>> exposes = new HashMap<>();

    public MARMPL1(){}

    public Boolean deploy(String id, JSONArray require, JSONArray expose) {...}

    public Boolean undeploy(String id) {...}

    public Boolean bind(String id, JSONArray require) {...}

    public Boolean unbind(String id, JSONArray require) {...}

    public Boolean expose(String id, JSONArray expose) {...}
}
```

Figura 21. Visión general de las funciones del objeto MARM.

6.1.2 Creación de la imagen *Docker*

Para la creación de la imagen utilizaremos el método habitual para *Docker*. La explicación del proceso se sale de los límites de este trabajo, pero si que mostraremos la política que se ha seguido para el etiquetado de estas imágenes.

Para la realización de este trabajo se va a utilizar un repositorio privado que reside en una máquina del departamento del DSIC donde trabaja el tutor de este proyecto.

Como este repositorio es compartido, hay que seguir ciertas convenciones para una mejor organización y facilitar una posterior búsqueda de las imágenes.

Así pues la etiqueta deberá mostrar:

- URL de la máquina → `tambori.dsic.upv.es`
- Puerto asociado a *Docker registry* → `10050`
- Un nombre de repositorio descriptivo del proyecto → `emba1pack`
- Un nombre de la imagen → `erp`
- Un *tag* que será su versión → `1.0`

Una etiqueta de una imagen de nuestro proyecto será entonces:

```
tambori.dsic.upv.es:10050/emba1pack/erp:1.0
```

Una vez determinado como serán las etiquetas se irán subiendo a este repositorio privado que posteriormente utilizará *Kubernetes* para cargar las imágenes (figs. 22 y 23).

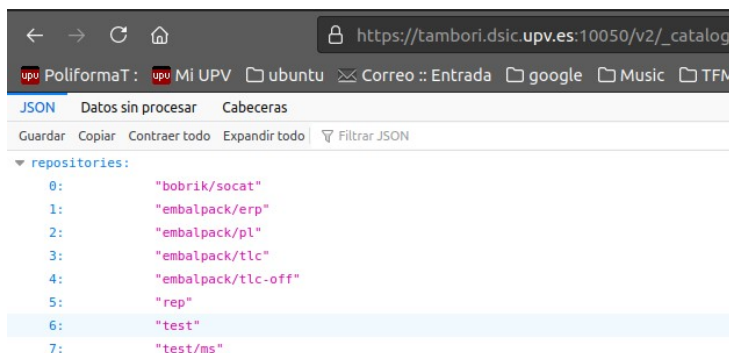


Figura 22. Lista de los repositorios creados en *Docker registry*.



Figura 23. Detalle de las versiones subidas a `/emba1pack/erp`.

6.1.3 Archivos de creación de objetos

Sabemos por lo visto en 3.4.2 que *Kubernetes* utiliza archivos YAML para la creación de objetos – *Deployments*, *Services*, *ConfigMaps*, etc – a continuación mostraremos como se han formado y qué es lo que crean.

Para la implementación de nuestro escenario crearemos *Deployment*, *Service* y *Secret*. *Deployment* cargará la imagen del ARM, creará el *Pod* y fijará el *ReplicaSet* a 1. *Service* dará acceso a la red a ARM y *Secret* nos proporcionará permisos para acceder a los repositorios *Docker*.

6.1.3.1 Deployment

A continuación se muestra el archivo de creación de un *Deployment*, que usaremos para resaltar las entradas más interesantes (fig. 24).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: erp-dpl
  namespace: embalpack
  labels:
    app: erp
    type: DevOps
spec:
  template:
    metadata:
      name: erp-pod
      labels:
        app: erp
        type: DevOps
    spec:
      containers:
        - name: erp-container
          image: tambori.dsic.upv.es:10050/embalpack/erp:1.0
          ports:
            - name: port8080
              containerPort: 8080
            - name: port8081
              containerPort: 8081
            - name: port8082
              containerPort: 8082
          imagePullSecrets:
            - name: regcred
      replicas: 1
      selector:
        matchLabels:
          type: DevOps
```

Figura 24. Archivo de creación de un Deployment para ARM ERP.

Donde:

metadata: Datos propios del *Deployment*.

spec: Datos del pod.

- **labels:** Especial relevancia de esta entrada porque será la que identifique unívocamente al *Pod*, como se puede observar esta entrada está en los datos del *Pod*, en las del *Deployment*, y más abajo en **selector**.
- **containers:** Sección que determina los datos de la imagen y cómo se va a configurar el contenedor.
 - **image:** Etiqueta de la imagen que cargará desde el repositorio.
 - **ports:** Puertos abiertos en el contenedor.
- **imagePullSecrets:** *Secret* que consultará para obtener acceso a *Docker registry*.

replicas: Número de réplicas que deberán estar siempre en funcionamiento.

selector: Entrada que utilizará *Service* para identificar al *Pod* al que conectarse, esto lo hace a través de **type** en **matchLabels**. Si observamos este **type** coincide con la entrada en **labels**.

6.1.3.2 Service

Seguiremos ahora con el archivo de creación de un *Service* mostrando un archivo donde igualmente hablaremos de las entradas (*fig. 25*).

```
apiVersion: v1
kind: Service
metadata:
  name: erp-ars-svc
spec:
  type: NodePort
  ports:
    - targetPort: 8080
      port: 30001
      nodePort: 30001
  selector:
    app: erp
    type: DevOps
```

Figura 25. Archivo de creación de un *Service nodePort* para ARM ERP.

Donde:

metadata: Datos propios del *Service*.

spec: Configuración del servicio.

- **type:** Existen tres tipos de servicio, *NodePort* que da acceso desde el exterior del clúster, *ClusterIP* que utiliza la red interna del clúster y *LoadBalancer* que se utiliza de proxy inverso para el reparto de carga.
- **ports:** Sección que determina los puertos que se van a utilizar.
 - **targetPort:** Puerto que utiliza el *Pod*.
 - **port:** Puerto que utilizará en la red interna del clúster.
 - **Nodeport:** Puerto que expondrá al exterior del clúster.
- **selector:** Entrada que utilizará el *Service* para saber el *Pod* al que debe servir. Si observamos **type** vemos que hace *matching* con el **type** del **selector** del *Deployment* y con la entrada **app** y **type** de **labels** en el *Pod*.

6.1.3.3 Secret

Para acceder al repositorio privado de *Docker* necesitamos unas credenciales como son un usuario y una contraseña. Estas credenciales deberá pues usarlas *Minikube* para poder descargar las imágenes del repositorio.

Kubernetes ofrece a través de los *Secrets* un acceso transparente a los repositorios al automatizar la introducción de estas credenciales.

Cuando nos identificamos en un repositorio privado, *Docker* guarda por defecto un archivo JSON con la dirección y las contraseñas de todos los repositorios a los que hayamos accedido. La dirección esta en texto claro, no así la contraseña que está cifrada (fig. 26).

```
"auths": {
  "192.168.1.18": {
    "auth": "bWlndWVsOjEzYnVmZmV0MTM="
  },
  "https://index.docker.io/v1/": {
    "auth": "dXB2dGZtMjAyMT0xMzZldDEz"
  },
  "tambori.dsic.upv.es:10050": {
    "auth": "YWRTaW46dGF0YW1p"
  }
}
```

Figura 26. Archivo JSON con las credenciales cifradas.

Con la lectura de este archivo *Kubernetes* es capaz de automatizar el proceso.

Para la creación del *Secret* tendremos que abrir la consola y utilizar `kubectl` (ver 3.4.3) para comunicarnos con el clúster.

El archivo con las credenciales está en `/root/.docker/config.json` de nuestra máquina. Al contrario que *Docker*, `kubectl` no se ejecuta como superusuario con lo que tendremos que copiar este archivo a nuestro HOME y darle permisos de lectura.

```
user@machine:~$ sudo cp /root/.docker/config.json .
```

Ahora crearemos el *Secret* con el siguiente comando:

```
user@machine:~$ kubectl create secret generic regcred --from /
file=.dockerconfigjson=config.json /
--type=Kubernetes.io/dockerconfigjson
```

El nombre `regcred` será el que figurará en el archivo de creación en la entrada `imagePullSecrets` del *Deployment* y que será el que utilice *Kubernetes*.

6.1.4 Ejecución del despliegue inicial

Para comenzar debemos crear un *Namespace* (ver 3.1.4) con esto conseguimos un espacio donde trabajar que está aislado en cuanto a recursos del resto.

```
kubectl create namespace test-deploy
```

En este momento ya tenemos todo lo necesario para probar nuestra implementación. Hemos creado los ARM y están alojados en *Docker registry*, tenemos el *Secret* ya creado y los archivos de creación de objetos.

Los archivos de creación serán los que hemos utilizado de muestra en la explicación y son los siguientes:

- *Deployment* → `ERP_dpl.yaml`
- *Service* → `ERP_ARS_svc.yaml` para la interfaz ARM,
 - `ERP_PLWO_svc.yaml` para exponer una interfaz del microservicio.
 - `ERP_REWVO_svc.yaml` para exponer una interfaz del microservicio.

Como no está todavía automatizado usaremos `kubectl` y *Postman* (ver 2.2.2).

Secuencia de comandos en `kubectl`:

1. `kubectl -n test-deploy create -f ERP_dpl.yaml`

```
2. kubectl -n test-deploy create -f ERP_ARS_svc.yaml
```

Con esto ya tenemos desplegado el ARM y su servicio en *Kubernetes*. Una de las características más atractivas de *Minikube* es que dispone de un *dashboard* que se muestra a través de web y que aquí usaremos para mostrar los resultados (figs. 27 y 28).

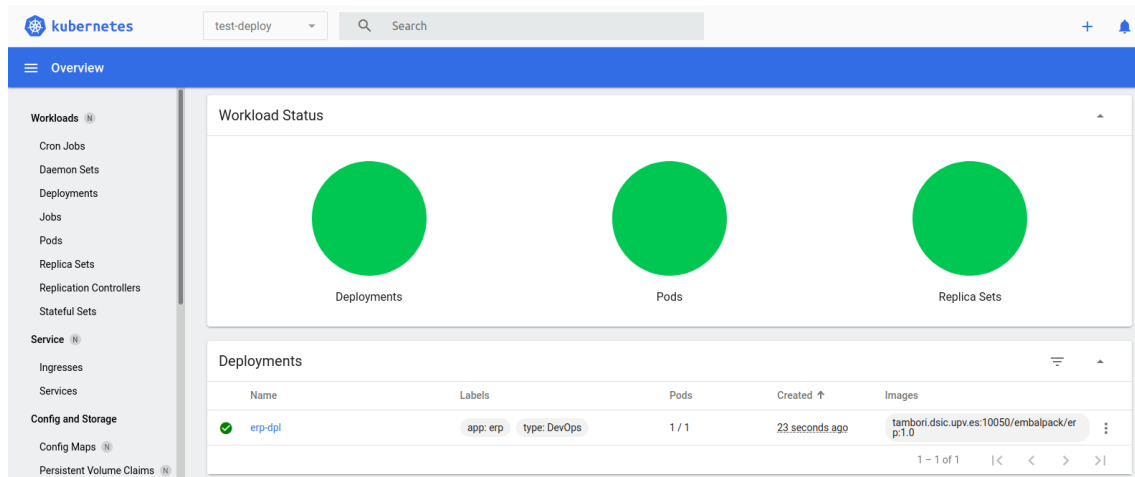


Figura 27. Dashboard de Minikube donde vemos el estado de los objetos y Deployment.

Ahora obtendremos la URL de acceso al ARM desde el exterior, esto se hace a través de *Service* que recordemos hemos configurado como *NodePort*.

```
minikube service erp-ars-svc --url
```

Donde `erp-ars-svc` es el nombre que le hemos dado al servicio en el fichero de creación.

Tenemos una respuesta de esta forma:

```
http://192.168.99.111:30001
```

Observamos que el puerto es el que le hemos indicado en la configuración.

Operación **Expose**:

1. Creamos el *Service* en *Minikube*

```
kubectl -n test-deploy create -f ERP_PLW0_svc.yaml
```

2. Ahora utilizando la URL y el API REST creado, usaremos *Postman* para configurar el microservicio a través de ARM.

Método → PUT

URL → `http://192.168.99.111:30001/ARM/ERP`

Cuerpo del texto →

```
{ "action": "expose", "require": [], "expose": { "Interface": "PLW0", "Available": true } }
```



}

Comprobamos el resultado (fig. 28).



Figura 28. Respuesta de ARM a la operación Expose.

Vemos el resultado en Minikube (fig. 29).

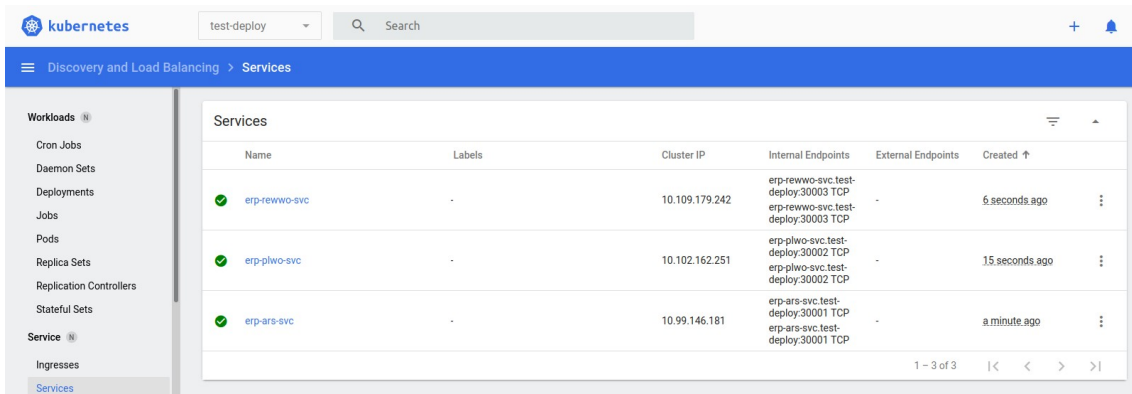


Figura 29. Dashboard de Minikube donde vemos los Services creados. Observamos los nombres DNS que se usan en la red interna.

Operación **Bind**:

Método → PUT

URL → http://192.168.99.111:30001/ARM/ERP

Cuerpo del texto →

```

{"action":"bind", "require":[{"id":"TLC", "url":"http://tlc1-svc.test-deploy:30092/TLC/TLC1"}],
"expose":[]
}
    
```

Comprobamos el resultado (fig. 30).



Figura 30. Respuesta de ARM a la operación Bind.

Operación **Unbind**:

Método → PUT

URL → `http://192.168.99.111:30001/ARM/ERP`

Cuerpo del texto →

```
{"action":"unbind", "require":[], "expose":[]}
```

Comprobamos el resultado (*fig. 31*):



Figura 31. Respuesta de ARM a la operación Unbind.

Por último realizaremos la operación `undeploy` sobre *Kubernetes* esto supone el borrado de todos los componentes desplegados.

1. `kubectl -n test-deploy delete deployment erp-dpl`
2. `kubectl -n test-deploy delete service erp-ars-svc`
3. `kubectl -n test-deploy delete service erp-plwo-svc`

Con esto habremos concluido con la prueba de funcionamiento de un despliegue inicial no automatizado.

6.2 Implementación del *Efactor* y la *Sonda*

Para continuar con la construcción de nuestra solución pasaremos a implementar los dos componentes que serán una extensión del bucle de control, el *Efactor* y la *Sonda*.

6.2.1. Implementación del *Efactor*

Este componente se implementa como un microservicio que expone una API REST para comunicarse con él. De esta manera la implementación será análoga a ARM pero con distinta funcionalidad.

Vamos a utilizar *Spring Boot* para la implementación, empezaremos creando el proyecto con *Maven* incluyendo las dependencias que creemos adecuadas. Recordemos que este componente debe utilizar *api-server* para comunicarse con *Kubernetes* (ver 4.4.1) por lo que deberemos incluir en las dependencias la librería *Java* para utilizar este *api-server*, esto es *io.Kubernetes* (*fig. 32*).



```
<dependency>
  <groupId>io.kubernetes</groupId>
  <artifactId>client-java-api</artifactId>
  <version>10.0.1</version>
</dependency>
<dependency>
  <groupId>io.kubernetes</groupId>
  <artifactId>client-java</artifactId>
  <version>10.0.1</version>
</dependency>
<dependency>
  <groupId>io.kubernetes</groupId>
  <artifactId>client-java-extended</artifactId>
  <version>10.0.1</version>
</dependency>
<dependency>
  <groupId>io.kubernetes</groupId>
  <artifactId>client-java-spring-integration</artifactId>
  <version>10.0.1</version>
</dependency>
<dependency>
  <groupId>io.kubernetes</groupId>
  <artifactId>client-java-proto</artifactId>
  <version>10.0.1</version>
</dependency>
```

Figura 32. Dependencias Maven para utilizar api-server de Kubernetes en el proyecto creado en Spring.

Utilizaremos esta estructura de paquetes y clases (fig. 33):

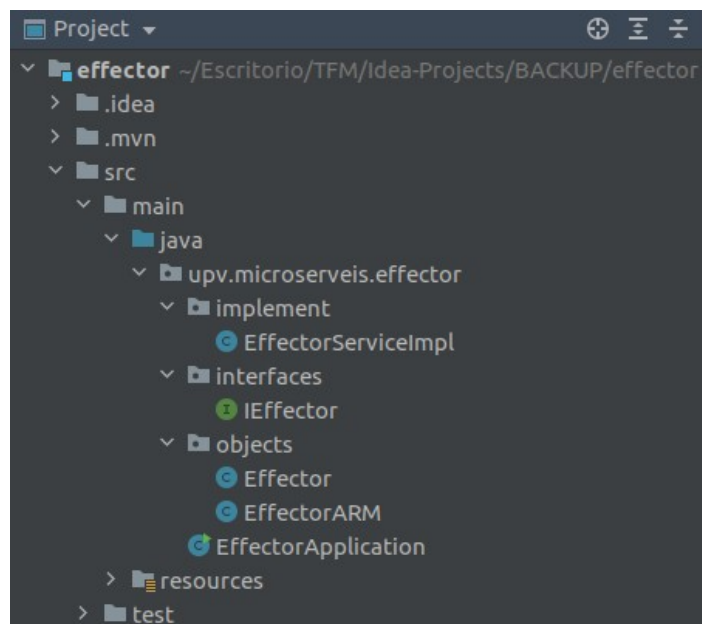


Figura 33. Estructura de paquetes y clases en Efactor.

Donde:

- El paquete `interfaces` contiene la clase que reciben los datos, aquí se declara la URL que se tiene que utilizar para enviarle las órdenes (fig. 34). Servirá como *endpoint* del componente.

```
package upv.microservis.effector.interfaces;

import ...

public interface IEffector {
    @GetMapping(value = "/hello")
    String sayHello();

    @PutMapping(value = "/ARM", consumes = "application/json", produces = "application/json")
    ResponseEntity<String> dataReception(@RequestBody String objson) throws Exception;
}
```

Figura 34. Declaración de la interfaz y la URL para API REST.

- El paquete `implement` contiene la clase que implementa las funciones de esta interfaz. Aquí se procesarán los mensajes JSON que contienen la información, p.e Deploy de ERP, o Expose la interfaz I de ERP (fig. 35).

```
@Override
public ResponseEntity<String> dataReception(String objson) throws Exception {
    Effector effector = EffectorApplication.effector;
    ResponseEntity<String> respuesta = ResponseEntity.badRequest().build();
    JSONObject requestValue = new JSONObject(objson);
    String action = requestValue.getString( key: "action");
    String name = requestValue.getString( key: "type");
    effectorARM = new EffectorARM();
    String interf = "";
    switch (action) {
        case "deploy":
            if (effector.deployment(name, requestValue)) {...}
            break;

        case "undeploy":
            if (effector.undeployment(name)) {...}
            break;

        case "bind":
            interf = requestValue.getString( key: "interface");
            if (effectorARM.bind(name, interf, requestValue)) {...}
            break;

        case "unbind":
            interf = requestValue.getString( key: "interface");
            if (effectorARM.unbind(name, interf)) {...}
            break;

        case "expose":
            interf = requestValue.getString( key: "interface");
            if (effectorARM.expose(name, interf)) {...}
            break;
    }
}
```

Figura 35. Fracción de código de la implementación de la función declarada en la interfaz.

- El paquete `object` contiene las clases que definen los objetos que se van a crear. En este paquete hay dos clases, una que corresponde al objeto `EffectorARM` que se encarga de la parte de los ARM y `Effector` que se encarga de la parte de `Kubernetes`. En el objeto `EffectorARM` es donde residen las funciones de que actúan sobre ARM. En

Effector es donde se establece la comunicación con *Kubernetes*, al igual que `kubectl` esto lo hace leyendo el archivo llamado 'config' en el directorio `$HOME/.kube` (ver 3.4.3) (*fig. 36*).

```
public Effector(String id) {
    this.id = id;
    try {
        apiClient = ClientBuilder.kubeconfig(KubeConfig.loadKubeConfig(new FileReader(kubeConfigPath))).build();
        Configuration.setDefaultApiClient(apiClient);
    } catch (IOException e) {
        e.printStackTrace();
        System.out.println("Configuration failed");
    }
    effectorARM = new EffectorARM();
}
```

Figura 36. Constructor en *Effector* donde se establece la conexión con *Kubernetes* .

6.2.2 Implementación de la *Sonda*

Este componente se va implementar como un microservicio que expone una API REST para enviarle la orden de arranque y establecer el objetivo. Una de las características de REST es que la respuesta es síncrona, esto quiere decir que hasta que el servidor no realice la operación demandada el cliente quedará a la espera.

Si lanzamos un proceso que mida periódicamente el estado del dispositivo, tanto el servidor como el cliente se quedarán suspendidos hasta la finalización de dicho proceso. Esto sería catastrófico para el funcionamiento del sistema, ya que uno de los requisitos es la medición continua.

La solución es una API REST asíncrona donde el servidor lanza el proceso de medición e inmediatamente después recupera el control de la ejecución, contestando al cliente y cerrando así la secuencia de eventos. El servidor queda entonces a la espera de nuevas peticiones. Para ello lanzaremos un hilo de ejecución por cada petición de arranque y establecimiento de objetivo. La ejecución de estos hilos no finalizará hasta que no se apague la *Sonda*.

Para implementar la *Sonda*, ya que es otro microservicio, lo construiremos de la misma forma que un ARM. Empezaremos creando el proyecto con *Gradle* incluyendo las dependencias que creemos son las adecuadas. Este componente debe reportar sus mediciones al monitor de MAPE-K utilizando colas de mensajería (ver 5.3.3), en concreto utilizaremos *MQTT paho* con lo que deberemos tenerlo en cuenta (*fig. 37*).

```
dependencies {
    compile 'org.eclipse.paho:org.eclipse.paho.client.mqttv3:1.0.2'
    compile 'org.eclipse.paho:org.eclipse.paho.android.service:1.0.2'
    implementation 'org.json:json:20201115'
    implementation 'org.springframework.boot:spring-boot-starter-validation'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    implementation 'org.springframework.cloud:spring-cloud-stream'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

Figura 37. Dependencias en el proyecto Gradle creado en Spring para Sonda.

Utilizaremos esta estructura de paquetes y clases (fig. 38):

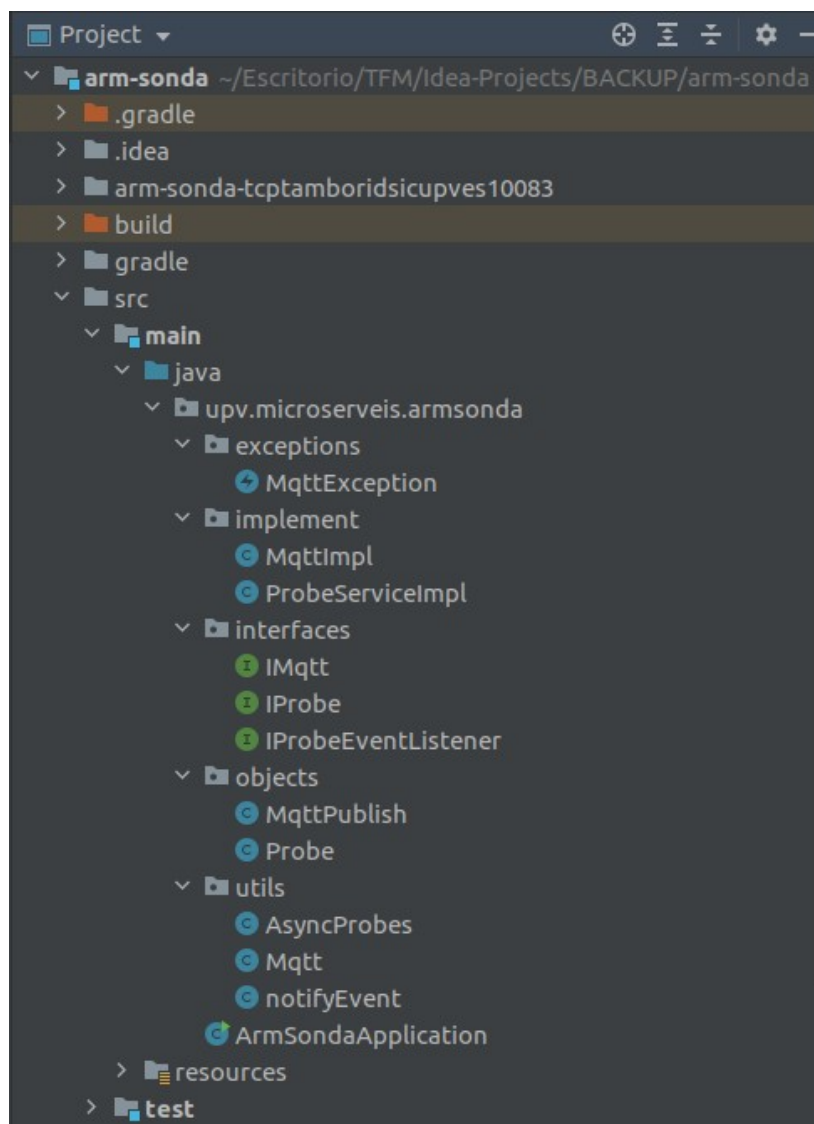


Figura 38. Estructura de paquetes y clases en Sonda.

Donde:

- El paquete `interfaces` contiene 3 clases:
 - `IProbe` → Clase que recibe los datos, aquí se declara la URL que tiene que utilizar el API REST de la *Sonda* (fig. 40). Servirá como endpoint del componente.
 - `IMqtt` → Clase que se comunica con el monitor via REST, esto es una alternativa a MQTT, por si el *broker* fallara.
 - `IProbeEventListener` → Interfaz que define el método que ejecutará cada uno de los hilos lanzados, con cada una de las mediciones.

```
package upv.microservels.armsonda.interfaces;

import ..

public interface IProbe {
    @GetMapping(value = "/status/{id}", produces = "application/json")
    Probe getStatus(@PathVariable String id);

    @GetMapping(value = "/hello")
    String sayHello();

    @PostMapping(value = "/start", consumes = "application/json", produces = "application/json")
    ResponseEntity<String> startProbe(@RequestBody String objson);
}
```

Figura 40. Declaración de la interfaz y la URL para API REST de Sonda.

- El paquete `implement` contiene las clases que implementan las funciones de esas interfaces. Aquí se procesarán los mensajes JSON que contienen la información necesaria para la creación de la *Sonda* y el dispositivo objetivo.

- El paquete `object` contiene las clases que definen los objetos que se van a crear.

- `MqttPublish` → Creación de una instancia cliente de MQTT.
- `Probe` → Creación de una instancia del objeto *Sonda*.

- El paquete `utils` contiene las clases que implementan la creación de los hilos de ejecución, así como los métodos MQTT.

- `Mqtt` → Conexión con el *broker* de mensajería MQTT.
- `AsyncProbes` → Creación del hilo de ejecución de una *Sonda* sobre un dispositivo objetivo (fig. 41).
- `notifyEvent` → Envío del mensaje con el estado del dispositivo al monitor (fig. 42).

```
// Asynchronous task
public void runProbe (String targetId, String targetUrl) {

    new Thread(new Runnable() {

        public void run()
        {
            String status="";
            RestTemplate restTemplate = new RestTemplate();
            if (mListener != null) {
                while(true) {
                    try {
                        Thread.sleep( millis: 2000);
                        status = restTemplate.getForEntity(targetUrl, String.class).getBody();

                        JSONObject data = new JSONObject(status);

                        if(data.getString( key: "status").equals("UP")) {
                            System.out.println("Servicio"+ targetId + " activo");
                            mListener.onProbeEvent(targetId, status: "H");
                        }
                        if(data.getString( key: "status").equals("DOWN")) {

                            mListener.onProbeEvent(targetId, status: "U");
                        }

                    } catch (InterruptedException e) {...} catch (Exception ex) {
                        System.out.println("Servicio"+ targetId + " caído");
                        mListener.onProbeEvent(targetId, status: "D");
                    }
                }
            }
        }
    })
}
```

Figura 41. Código del lanzamiento de hilos de ejecución y el tratamiento de las respuestas del dispositivo.

```
@Override
public void onProbeEvent(String micro, String status) {

    JSONObject aux = new JSONObject();
    aux.put("message", "{\n" +
        "    \"health-check\": \"\"+ status + "\",\n" +
        "    \"microservice\": \"\"+ micro + "\",\n" +
        "    }");
    aux.put("retained", "True");
    aux.put("qos", "0");
    aux.put("topic", "ambalpack/monitors/health");

    try {
        this.publishMessageEvent(aux);
    } catch (MqttException e) {
        e.printStackTrace();
    }
}

public void publishMessageEvent(JSONObject message) throws org.eclipse.paho.client.mqttv3.MqttException {

    MqttMessage mqttMessage = new MqttMessage(message.getString( key: "message").getBytes());
    mqttMessage.setQos(message.getInt( key: "qos"));
    mqttMessage.setRetained(message.getBoolean( key: "retained"));
    Mqtt.getInstance().publish(message.getString( key: "topic"), mqttMessage);
}
}
```

Figura 42. Fracción de código de la construcción y envío del mensaje a FADA.

- El paquete `exceptions` contiene la clase que gestionará las excepciones que se produzcan con la ejecución de las colas de mensajería MQTT.



6.2.3 Test de *Efactor* y *Sonda*

A continuación realizaremos la prueba de funcionamiento de estos componentes. Comprobaremos en primer lugar el *Efactor* para ello haremos una simulación de automatización mediante un *shell script* que irá lanzando peticiones al componente. Observaremos los resultados por consola y el *dashboard* de *Minikube*. Seguidamente probaremos la *Sonda* comprobando si responde al estado de un determinado dispositivo.

6.2.3.1 Test de *Efactor*

Para la realización del test y posterior uso en la implementación del caso de estudio, estos componentes se han subido al servidor del DSIC `tambori.dsic.upv.es`, esto es debido a que el *Efactor* debe trabajar en la misma máquina donde se encuentra *Minikube*, por lo tanto trabajarán en local dentro del servidor. La *Sonda* trabajará de forma análoga controlando los dispositivos porque el escenario del caso de estudio se va a desplegar en el servidor antes mencionado .

Escribimos un *script* con las peticiones que posteriormente ejecutaremos (*fig. 43*)

```
#!/bin/bash
curl -X PUT -H "Content-Type: application/json" -d '{"type":"ERP","action":"deploy"}' http://tambori.dsic.upv.es:10060/ARM
curl -X PUT -H "Content-Type: application/json" -d '{"type":"PL1","action":"deploy"}' http://tambori.dsic.upv.es:10060/ARM
curl -X PUT -H "Content-Type: application/json" -d '{"type":"TLC1","action":"deploy"}' http://tambori.dsic.upv.es:10060/ARM

curl -X PUT -H "Content-Type: application/json" -d '{"type":"ERP","action":"expose","interface":"PLWO"}' http://tambori.dsic.upv.es:10060/ARM
curl -X PUT -H "Content-Type: application/json" -d '{"type":"ERP","action":"expose","interface":"REWWO"}' http://tambori.dsic.upv.es:10060/ARM
curl -X PUT -H "Content-Type: application/json" -d '{"type":"PL1","action":"expose","interface":"PL"}' http://tambori.dsic.upv.es:10060/ARM
curl -X PUT -H "Content-Type: application/json" -d '{"type":"TLC1","action":"expose","interface":"TLC"}' http://tambori.dsic.upv.es:10060/ARM
```

Figura 43. Shell script para realizar el test del *Efactor*.

Vemos ahora el resultado obtenido por consola desde nuestra máquina (*fig. 44*):

```
r2d3@c3p0:~/Escritorio/TFM/TXT-JAVA-BASH$ ./test-efactor.sh
Deployment Created
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: ERPDeployment Created
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: PL1Deployment Created
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: TLC1Expose PLWO action done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: ERPEXpose REWWO action done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: ERPEXpose PL action done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: PL1Expose TLC action done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: TLC1
```

Figura 44. Salida por consola en local de la respuesta del *Efactor*.

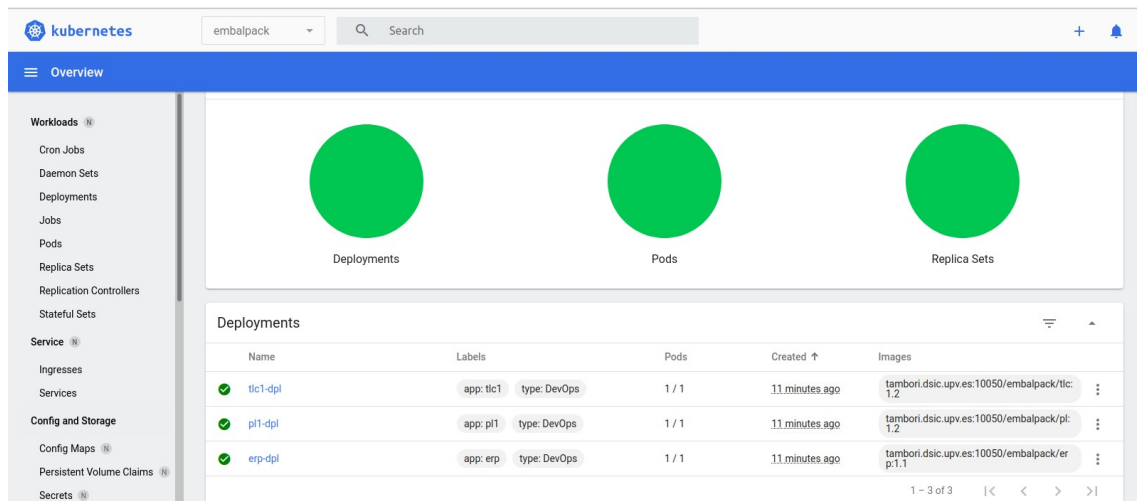
Y el resultado del *Efactor* en el servidor (fig. 45):

```
2021-06-22 17:25:42.237 INFO 9732 --- [      main] o.w.effector.EffectorApplication : Started EffectorApplication in 2.173 seconds (JVM running for 3.799)
2021-06-22 17:27:10.277 INFO 9732 --- [io-10060-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-06-22 17:27:10.277 INFO 9732 --- [io-10060-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-06-22 17:27:10.278 INFO 9732 --- [io-10060-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
Deployment ERP Created!
Service ERP_ARS Created!
Service URL: http://192.168.99.102:30001/ARM/ERP
Deploy action done
Deployment PL1 Created!
Service PL1_ARS Created!
Service URL: http://192.168.99.102:30020/ARM/PL1
Deploy action done
Deployment TLC1 Created!
Service TLC1_ARS Created!
Service URL: http://192.168.99.102:30025/ARM/TLC1
Deploy action done
Service ERP_PLW0 Created!
Service URL: http://192.168.99.102:30001/ARM/ERP
Evento enviado
Expose PLW0 action done
Service ERP_REW0 Created!
Service URL: http://192.168.99.102:30001/ARM/ERP
Evento enviado
Expose REW0 action done
Service PL1_PL Created!
Service URL: http://192.168.99.102:30020/ARM/PL1
Evento enviado
Expose PL action done
Service TLC1_TLC Created!
Service URL: http://192.168.99.102:30025/ARM/TLC1
Evento enviado
Expose TLC action done
```

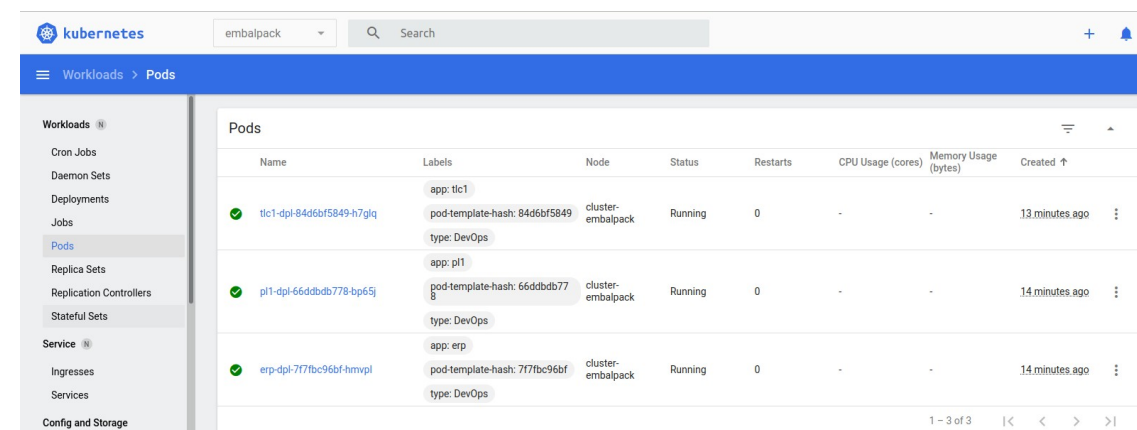
Figura 45. Salida por consola de la respuesta del Efactor en tambori.dsic.upv.es.

A continuación algunas capturas del dashboard de Minikube:

Deployments



Pods



Services

Name	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created ↑
tlc1-trlc-svc	-	10.109.133.118	tlc1-trlc-svc.embalpack:30026 TCP	-	12.minutes.ago
pl1-pl-svc	-	10.104.121.186	pl1-pl-svc.embalpack:30021 TCP	-	12.minutes.ago
erp-rewwo-svc	-	10.102.85.38	erp-rewwo-svc.embalpack:30003 TCP	-	12.minutes.ago
erp-plwo-svc	-	10.97.54.227	erp-plwo-svc.embalpack:30002 TCP	-	12.minutes.ago
erp-plwo-svc	-	10.97.54.227	erp-plwo-svc.embalpack:30002 TCP	-	12.minutes.ago
tlc1-ars-svc	-	10.104.203.248	tlc1-ars-svc.embalpack:30025 TCP	-	13.minutes.ago
pl1-ars-svc	-	10.110.110.143	pl1-ars-svc.embalpack:30020 TCP	-	13.minutes.ago
erp-ars-svc	-	10.107.47.229	erp-ars-svc.embalpack:30001 TCP	-	13.minutes.ago

A continuación lanzaremos un script que realizará la operación undeploy y que borrará *Deployments* y *Services*.

Vemos ahora el resultado obtenido por consola en nuestra máquina:

```
r2d3@c3p0:~/Escritorio/TFM/TXT-JAVA-BASH$ ./test-eff-undpl.sh
Undeployment done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: ERPUndeployment done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: PL1Undeployment done
Effector ID: 3c208c16-a9fc-41ef-b6c9-182a649e4ecd
Microservei: TLC1
```

Y el resultado del *Effector* en el servidor:

```
Service URL: http://192.168.99.102:30001/ARM/ERP
Evento enviado
Service erp-ars Deleted!
Service removed:
- ERP
Service erp-plwo Deleted!
Service removed:
- ERP
Service erp-rewwo Deleted!
Service removed:
- ERP
Undeploy action done
```

```
Service URL: http://192.168.99.102:30020/ARM/PL1
Evento enviado
Service pl1-ars Deleted!
Service removed:
- PL1
Service pl1-pl Deleted!
Service removed:
- PL1
Undeploy action done
Service URL: http://192.168.99.102:30025/ARM/TLC1
Evento enviado
Service tlc1-ars Deleted!
Service removed:
- TLC1
Service tlc1-trlc Deleted!
Service removed:
- TLC1
Undeploy action done
```



6.2.3.1 Test de la Sonda

En este test arrancaremos cuatro microservicios y la *Sonda*. Después mandaremos los dispositivos objetivo a través de API REST. Crearemos un *script* para realizar estas tareas (fig. 46).

```
#!/bin/bash
java -jar arm-sonda-0.0.1-SNAPSHOT.jar 10090 &
sleep 5
java -jar DevPL1-0.0.1-SNAPSHOT.jar DevPL1 10061 &
sleep 5
}
ava -jar DevTL1-0.0.1-SNAPSHOT.jar DevTL1 10067 &
sleep 5
java -jar DevPL1-0.0.1-SNAPSHOT.jar DevPL2 10062 &
sleep 5
java -jar DevTL1-0.0.1-SNAPSHOT.jar DevTL2 10068 &
sleep 5
curl -X POST -H "Content-Type: application/json" -d '{"target":{"id":"DevPL1", "url":"http://localhost:10061/actuador/health"}}' http://localhost:10090/start
sleep 3
curl -X POST -H "Content-Type: application/json" -d '{"target":{"id":"DevTL1", "url":"http://localhost:10067/actuador/health"}}' http://localhost:10090/start
sleep 3
curl -X POST -H "Content-Type: application/json" -d '{"target":{"id":"DevPL2", "url":"http://localhost:10062/actuador/health"}}' http://localhost:10090/start
sleep 3
curl -X POST -H "Content-Type: application/json" -d '{"target":{"id":"DevTL2", "url":"http://localhost:10068/actuador/health"}}' http://localhost:10090/start
sleep 3
```

Figura 46. Shell script para realizar el test de la Sonda.

Comprobamos el funcionamiento consultando la salida por consola de la *Sonda*:

```
{"id": "DevTL2", "url": "http://localhost:10068/actuador/health"}
{"target": {"id": "DevTL2", "url": "http://localhost:10068/actuador/health"}}-----
Sonda creada
Conectada a: {"target": {"id": "DevTL2", "url": "http://localhost:10068/actuador/health"}}
```

Sonda en ejecución:

```
ServicioDevPL1 activo
ServicioDevTL2 activo
ServicioDevTL1 activo
ServicioDevPL2 activo
```

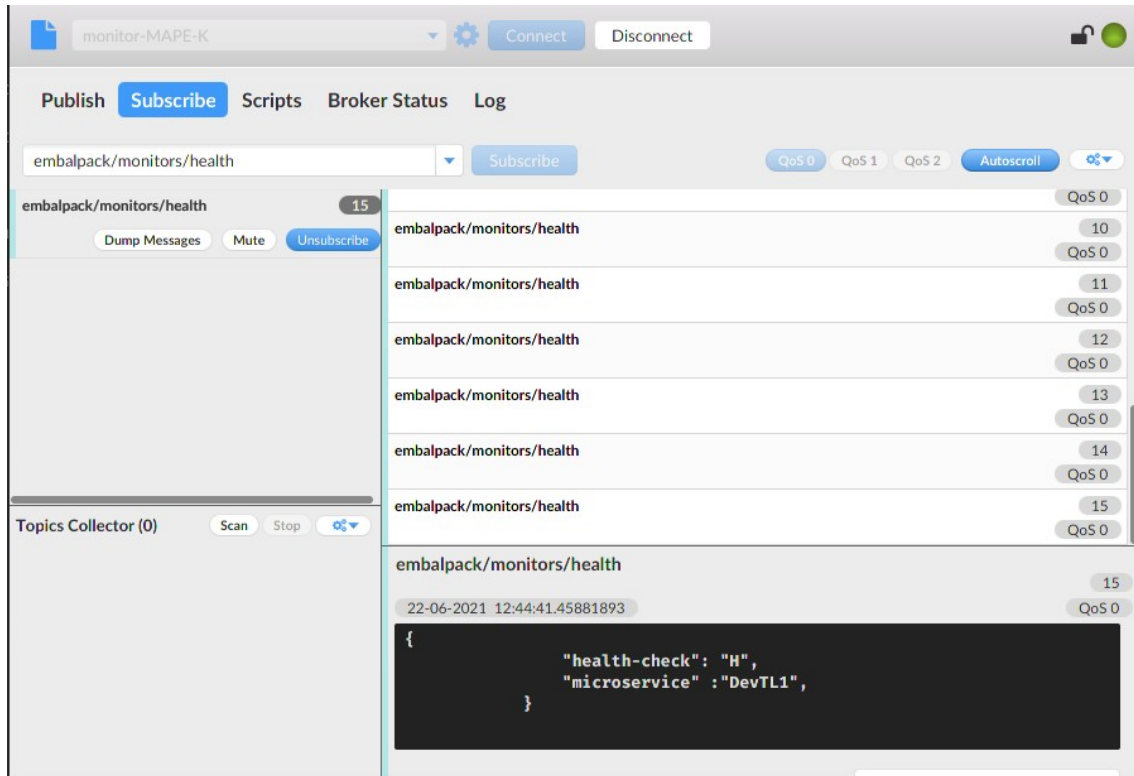
Paramos DevPL1:

```
ServicioDevPL1 caído
ServicioDevTL2 activo
ServicioDevTL1 activo
ServicioDevPL2 activo
```

Paramos DevTL1:

```
ServicioDevPL1 caído
ServicioDevTL2 activo
ServicioDevTL1 caído
ServicioDevPL2 activo
```

Vemos el envío de los mensajes por MQTT utilizando MQTT.fx. En esta ocasión reporta un 'HEALTHY' del microservicio DevTL1:



Cuando paramos DevPL1 el mensaje cambia a 'DEAD':



Con esto habremos concluido con la implementación de todos los componentes pertenecientes a nuestra solución. A continuación realizaremos la implementación en el escenario real motivo del caso de estudio, Embalpack.

6.3 Implementación del caso de estudio. Embalpack.

Como se ha descrito en 3.5, Embalpack es una fábrica en la que se producen cantoneras de cartón. Esta fábrica dispone, entre otros, de una serie de microservicios que controlan los recursos tales como equipos y operarios, o controladores que gestionan los semáforos de las cadenas de producción. Para la realización de la implementación final vamos a utilizar una pequeña porción de toda la infraestructura. Si la implementación es exitosa se puede trasladar fácilmente al resto del sistema.

6.3.1 Componentes participantes

6.3.1.1 Semáforos de producción. DevTL.

Los semáforos son equipos físicos de bajo rendimiento que muestran al operario de una manera rápida y visual el estado del ritmo de producción – en relación al esperado – . Así un color verde indica un buen ritmo y la falta de incidencias, el amarillo es un valor intermedio y el rojo muestra que el ritmo de producción es bajo, con lo que el operario deberá actuar en consecuencia.

El servicio ligado a este dispositivo físico, que se ejecuta en módulos de cómputo externos – Raspberry Pi – se conecta directamente al semáforo a través de sus interfaces de comunicación físicas. El servicio recibe del controlador IoT los datos que mostrará el semáforo.

Este componente se denomina DevTL y expone una interfaz de comunicación DevTL y requiere de dos enlaces para su funcionamiento Digital Twins y ThingsREG (fig. 47).

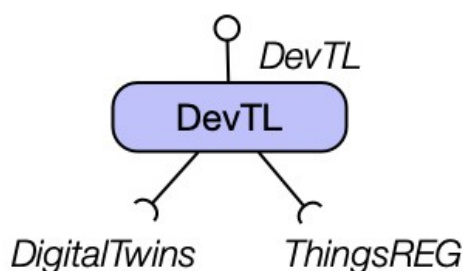


Figura 47. Notación para DevTL.

El servicio se implementa de igual forma que los otros microservicios, este obviamente no es un ARM no será pues auto-adaptativo.

6.3.1.2 Máquinas de producción. DevPL.

Otro de los dispositivos físicos presentes en el sistema es el de las máquinas de producción. Estos dispositivos también están ligados a un servicio que ofrece una interfaz física con la línea de producción. Permite extraer datos/indicadores físicos de producción – velocidad de producción actual/real, paradas de máquina, configuración de longitud de cortes, número de cortes realizados, etc. – así como cambiar la configuración de la máquina – velocidad de producción y longitud de corte – . Genera información sobre ubicación /localización de recursos que procesa.

Este servicio se denomina DevPL y expone una interfaz de comunicación DevPL y requiere de dos enlaces para su funcionamiento Digital Twins y ThingsREG (fig. 48).

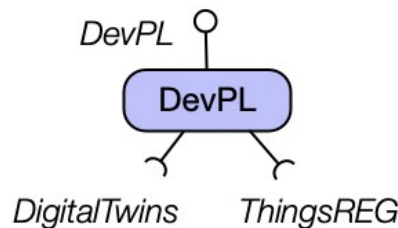


Figura 48. Notación para DevPL.

El servicio se implementa de igual forma que los otros microservicios, este tampoco es un ARM.

6.3.1.3 Extensión del sistema productivo. PL y TLC

En el sistema hay servicios que extienden procesos productivos, recolectando y procesando datos, o proporcionando interfaces de acceso a los dispositivos físicos.

Así los servicios PL están vinculados a las máquinas de producción y los TLC son controladores de semáforos.

El servicio PL expone una interfaz de comunicación PL y requiere de dos enlaces para su funcionamiento PLWO Y DevPL (fig. 49).

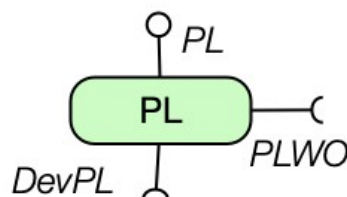


Figura 49. Notación para PL.

El servicio TLC expone una interfaz de comunicación TLC y requiere de tres enlaces para su funcionamiento PLWO, DevPL y DevTL (fig. 50).

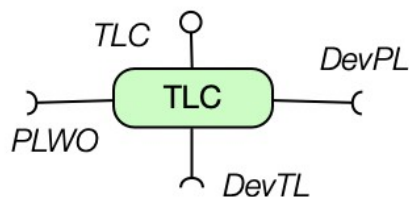


Figura 50. Notación para TLC.

Estos dos servicios serán encapsulados en un ARS/ARM, con lo que se podrán realizar las operaciones de adaptación sobre ellos.

6.3.1.4 Servicios de infraestructura. ERP de producción.

Este no es un microservicio, y quizás sea complicado argumentar que lo es. Pero, se ha desarrollado infraestructura para que se pueda conectar a la plataforma IoT a través de una API REST y comunicarse mediante MQTT con diferentes recursos y máquinas. Mantiene información sobre identificadores y tipos de recursos de la fábrica.

De igual manera que PL y TLC será encapsulado en un ARS/ARM.

Este componente expone dos interfaces de comunicación REWWO y PLWO y requiere de un enlace para su funcionamiento TLC (fig. 51).

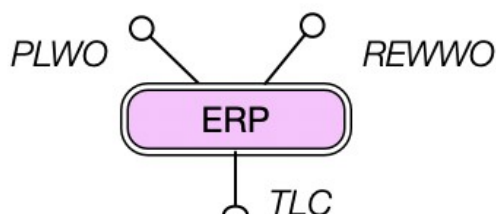


Figura 51. Notación para ERP.

6.3.1.5 Servicio auxiliar. TLCoffline.

Este servicio se ha desarrollado para esta implementación, actualmente no existe en el sistema. Requerirá de un enlace donde se conectará el dispositivo físico – DevTL – en caso de fallo del controlador TLC. Su puesta en marcha forma parte de uno de los procesos de adaptación.



El componente se implementa de forma análoga a PL, TLC y ERP. Será pues encapsulado en un ARS/ARM.

El servicio requiere de un enlace para su funcionamiento DevTL (fig. 52).

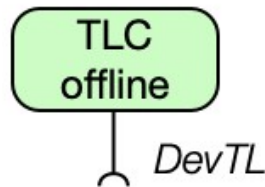


Figura 52. Notación para TLCOffline.

6.3.2 Escenario

En este apartado presentaremos un caso de uso de un sistema auto-adaptativo que opera sobre *Kubernetes*, mostraremos la configuración arquitectónica de un subconjunto de los microservicios de la solución ejecutándose sobre la infraestructura de Embalpack, y el escenario de adaptación en el que realizaremos la implementación final.

6.3.2.1 Configuración arquitectónica

Así pues vamos a emplear todo lo que hemos desarrollado hasta ahora. En este escenario tenemos lo siguiente:

- Bucle MAPE-K *FADA*
- *Minikube*
- Registro de contenedores *Docker registry*
- ARS/ARM
- *Efector*
- *Sonda*
- Dispositivos virtuales

En esta configuración *FADA* ejecutará los bucles de control, donde la *Sonda* reportará con mensajes 'health-check' al monitor del estado de los dispositivos. Si un dispositivo falla, *FADA* realizará la secuencia de adaptación MAPE y enviará las órdenes al *Efector*. Este componente actuará sobre *Minikube* y ARS/ARM. *Minikube* obtendrá las imágenes *Docker* de *Doker registry* (fig. 53).

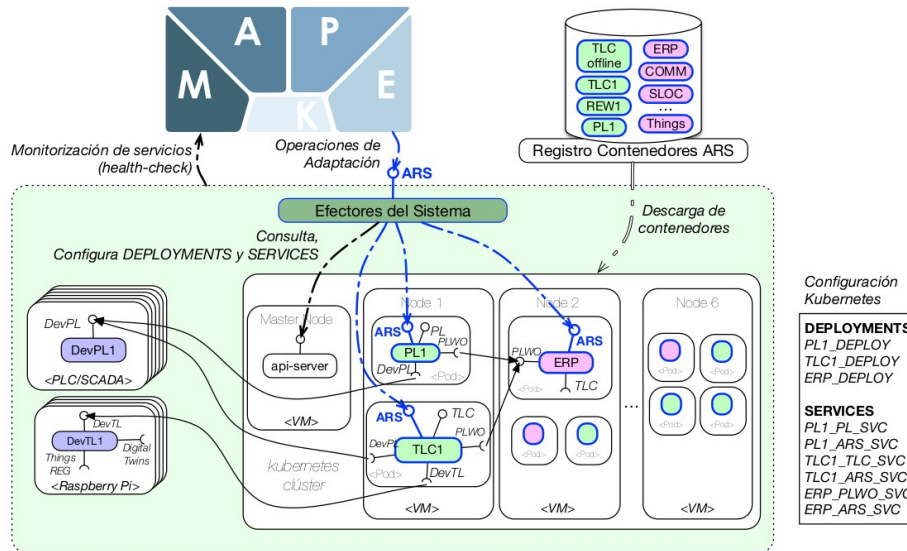


Figura 53. Configuración arquitectónica basada en ARS sobre Kubernetes [11].

6.3.2.1 Escenario de adaptación

Atendiendo a los microservicios diseñados para la fábrica y sus dependencias, existen situaciones en las que se requiere reconfigurar dinámicamente la arquitectura de la solución. Cada necesidad de reconfiguración se ha desarrollado como un escenario de adaptación que se ha implementado a través de una política de adaptación dentro el bucle de control. Estos escenarios se disparan cuando se detecta una condición sobre la infraestructura de microservicios – típicamente, debido a monitorizaciones de tipo health-check –, y para cada escenario, la política de adaptación propone un conjunto de operaciones de adaptación. Es muy importante tener en cuenta que estos servicios sólo son funcionales si todas sus interfaces son accesibles o pueden enlazarse a otro microservicio.

Los componentes de este escenario son los siguientes (fig. 54):

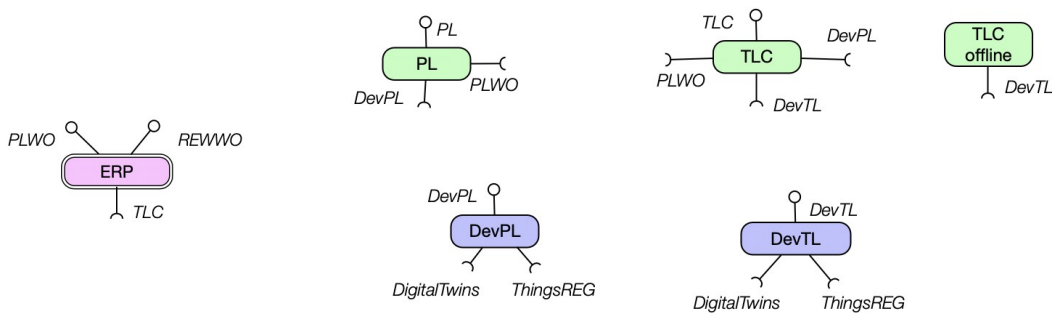
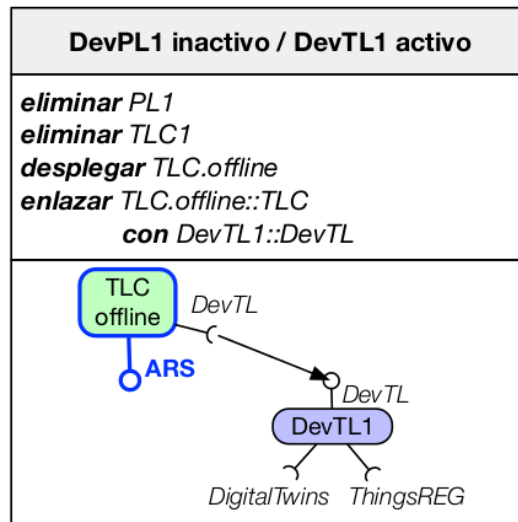


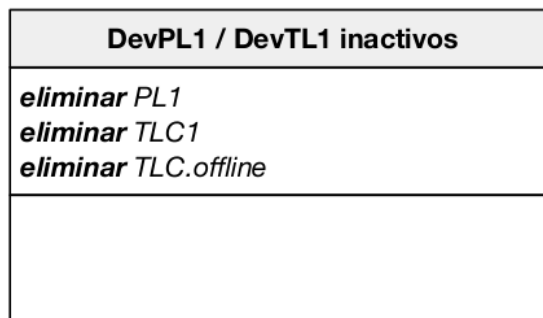
Figura 54. Servicios involucrados en el escenario de adaptación.



- Caso 2 de adaptación: DevTL esta activo y DevPL está inactivo.



- Caso 3 de adaptación: DevTL y DevPL están inactivos.



Hemos mostrado los escenarios relacionados con la disponibilidad – o no – de los servicios que se ejecutan embebidos en las máquinas – DevPL – y en los semáforos – DevTL – , aplicados sobre una línea de producción. En cada situación, el bucle de adaptación solicita la ejecución de las operaciones de adaptación asociadas a cada escenario, provocando la reconfiguración arquitectónica y estableciendo la configuración.

6.4 Test final

Para realizar este test vamos a utilizar todo lo desarrollado, utilizando parte de la infraestructura de Embalpack. Se va utilizar *FADA* como bucle de control, y para aproximarse lo más posible a la situación real en la fábrica se implementa DevPL sobre una raspBerry Pi. Así mismo se implementa una *Sonda* junto al DevPL en la misma ubicación. El resto de componentes no varían su ubicación.



6.4.1 Resultados

Ahora mostraremos los resultados obtenidos en la realización del test. Arrancaremos DevPL, la *Sonda*, *FADA* y el *Efactor*. Observaremos la configuraciones iniciales, posteriormente apagaremos DevPL y veremos un escenario de adaptación.

1 - Arranque de FADA (fig. 55):

```
osgi> 2021/06/26 15:17:29.663 INFO AdaptiveReadyComponent : [Knowledge] BINDING SERVICE ...
2021/06/26 15:17:29.669 INFO AdaptiveReadyComponent : [Knowledge] BINDING SERVICE ...
2021/06/26 15:17:29.908 INFO AdaptiveReadyComponent : [Knowledge] SETTING PARAMETER ...
2021/06/26 15:17:29.909 INFO AdaptiveReadyComponent : [Knowledge] DEPLOYING ...
2021/06/26 15:17:29.912 INFO AdaptiveReadyComponent : [KnowledgeModule] BINDING SERVICE ...
2021/06/26 15:17:29.916 INFO AdaptiveReadyComponent : [KnowledgeModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.916 INFO AdaptiveReadyComponent : [MonitoringModule] BINDING SERVICE ...
2021/06/26 15:17:29.920 INFO AdaptiveReadyComponent : [KnowledgeModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.921 INFO AdaptiveReadyComponent : [AnalyzingModule] BINDING SERVICE ...
2021/06/26 15:17:29.927 INFO AdaptiveReadyComponent : [AnalyzingModule] BINDING SERVICE ...
2021/06/26 15:17:29.927 INFO AdaptiveReadyComponent : [AnalyzingModule] DEPLOYING ...
2021/06/26 15:17:29.927 INFO AdaptiveReadyComponent : [AnalyzingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.930 INFO AdaptiveReadyComponent : [MonitoringModule] BINDING SERVICE ...
2021/06/26 15:17:29.931 INFO AdaptiveReadyComponent : [MonitoringModule] DEPLOYING ...
2021/06/26 15:17:29.935 INFO AdaptiveReadyComponent : [KnowledgeModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.936 INFO AdaptiveReadyComponent : [PlannerModule] BINDING SERVICE ...
2021/06/26 15:17:29.936 INFO AdaptiveReadyComponent : [PlannerModule] DEPLOYING ...
2021/06/26 15:17:29.937 INFO AdaptiveReadyComponent : [PlannerModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.937 INFO AdaptiveReadyComponent : [AnalyzingModule] BINDING SERVICE ...
2021/06/26 15:17:29.940 INFO AdaptiveReadyComponent : [KnowledgeModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.940 INFO AdaptiveReadyComponent : [ExecutingModule] BINDING SERVICE ...
2021/06/26 15:17:29.941 INFO AdaptiveReadyComponent : [ExecutingModule] DEPLOYING ...
2021/06/26 15:17:29.941 INFO AdaptiveReadyComponent : [ExecutingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:29.941 INFO AdaptiveReadyComponent : [PlannerModule] BINDING SERVICE ...
2021/06/26 15:17:29.947 INFO AdaptiveReadyComponent : [PlannerModule] BINDING SERVICE ...
2021/06/26 15:17:29.950 INFO AdaptiveReadyComponent : [ExecutingModule] BINDING SERVICE ...
2021/06/26 15:17:29.985 INFO AdaptiveReadyComponent : [ARM.SystemEffectors.Kubernetes] DEPLOYING ...
2021/06/26 15:17:29.985 INFO AdaptiveReadyComponent : [Executor] BINDING SERVICE ...
2021/06/26 15:17:40.173 INFO AdaptiveReadyComponent : [MonitoringModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.175 INFO AdaptiveReadyComponent : [self-configure-monitor] BINDING SERVICE ...
2021/06/26 15:17:40.176 INFO AdaptiveReadyComponent : [self-configure-monitor] DEPLOYING ...
2021/06/26 15:17:40.178 INFO AdaptiveReadyComponent : [MonitoringModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.179 INFO AdaptiveReadyComponent : [Microservice-HealthStatus-monitor] BINDING SERVICE ...
2021/06/26 15:17:40.179 INFO AdaptiveReadyComponent : [Microservice-HealthStatus-monitor] DEPLOYING ...
2021/06/26 15:17:40.279 INFO AdaptiveReadyComponent : [AnalyzingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.279 INFO AdaptiveReadyComponent : [self-configure-adaptationrule] BINDING SERVICE ...
2021/06/26 15:17:40.281 INFO AdaptiveReadyComponent : [self-configure-adaptationrule] DEPLOYING ...
2021/06/26 15:17:40.288 INFO AdaptiveReadyComponent : [AnalyzingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.288 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline1-adaptationrule] BINDING SERVICE ...
2021/06/26 15:17:40.289 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline1-adaptationrule] DEPLOYING ...
2021/06/26 15:17:40.290 INFO AdaptiveReadyComponent : [AnalyzingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.291 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline2-adaptationrule] BINDING SERVICE ...
2021/06/26 15:17:40.292 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline2-adaptationrule] DEPLOYING ...
2021/06/26 15:17:40.293 INFO AdaptiveReadyComponent : [AnalyzingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.293 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline3-adaptationrule] BINDING SERVICE ...
2021/06/26 15:17:40.294 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline3-adaptationrule] DEPLOYING ...
2021/06/26 15:17:40.295 INFO AdaptiveReadyComponent : [AnalyzingModule] GET SERVICE SUPPLY ...
2021/06/26 15:17:40.295 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline4-adaptationrule] BINDING SERVICE ...
2021/06/26 15:17:40.296 INFO AdaptiveReadyComponent : [microservice-healthstatus-productionline4-adaptationrule] DEPLOYING ...
2021/06/26 15:17:40.299 INFO AdaptiveReadyComponent : [self-configure-probe] BINDING SERVICE ...
2021/06/26 15:17:40.299 INFO AdaptiveReadyComponent : [self-configure-probe] DEPLOYING ...
2021/06/26 15:17:40.300 INFO AdaptiveReadyComponent : [Microservice-HealthStatus-Probe] BINDING SERVICE ...
2021/06/26 15:17:40.301 INFO AdaptiveReadyComponent : [Microservice-HealthStatus-Probe] DEPLOYING ...
```

Figura 55. Secuencia de encendido del bucle MAPE-K. Se puede apreciar qué sección del bucle está actuando en cada momento.

2 - Inicio de la configuración inicial – auto-configuración – (fig. 56).

```
2021/06/26 15:17:40.301 INFO Probe : (self-configure-probe) Reporting measure: factory-reset
2021/06/26 15:17:40.302 DEBUG Monitor : Received measure: factory-reset
2021/06/26 15:17:40.322 TRACE LoopResource : <Rule self-configure-adaptationrule> Asking for the next System Configuration
2021/06/26 15:17:40.395 TRACE LoopResource : <Rule self-configure-adaptationrule> Request Planning (3defee67975e44e79205545cf832b54a)
2021/06/26 15:17:40.386 INFO AnalyzingModule : >>> ## STARTING NEW ADAPTATION PROCESS (3defee67975e44e79205545cf832b54a) <<<
2021/06/26 15:17:40.421 INFO AdaptiveReadyHost : [kubernetes] DEPLOYING HOST
2021/06/26 15:17:40.423 INFO AdaptiveReadyHost : [kubernetes] DEPLOYING ERP CONTAINER
2021/06/26 15:17:40.423 INFO AdaptiveReadyContainer : [ERP] DEPLOYING ERP CONTAINER IN kubernetes HOST
2021/06/26 15:17:40.427 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>> <api-server> CREATE DEPLOYMENT: erp_deployment.embalpack
2021/06/26 15:17:40.427 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>> <api-server> CREATE SERVICE: erp-ars-svc.embalpack
2021/06/26 15:17:52.916 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>> <api-server> CREATE DEPLOYMENT: erp_deployment.embalpack (done)
2021/06/26 15:17:52.917 INFO AdaptiveReadyContainer : [ERP] DEPLOYING ERP MICROSERVICE
2021/06/26 15:17:52.921 INFO AdaptiveReadyMicroservice : [ERP] DEPLOYING ERP MICROSERVICE IN ERP CONTAINER
2021/06/26 15:17:52.922 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>> <api-server> CREATE SERVICE: erp-plwo-svc.embalpack
2021/06/26 15:18:00.233 INFO AdaptiveReadyMicroservice : [ERP] EXPOSING PLWO SERVICE THROUGH PLWO INTERFACE
2021/06/26 15:18:00.234 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>> <api-server> CREATE SERVICE: erp-rewwo-svc.embalpack
2021/06/26 15:18:07.655 INFO AdaptiveReadyMicroservice : [ERP] EXPOSING REWMO SERVICE THROUGH REWMO INTERFACE
2021/06/26 15:18:07.656 INFO AdaptiveReadyMicroservice : [ERP] DEFINING TLC REQUIRED SERVICE THROUGH TLC INTERFACE
2021/06/26 15:18:07.656 INFO ExecutorModule : Adaptation success. Updating the current configuration ...
2021/06/26 15:18:07.664 INFO AnalyzingModule : >>> ## ADAPTATION PROCESS ENDED (3defee67975e44e79205545cf832b54a) <<<
```

Figura 56. Auto-configuración inicial. Vemos como se despliega ERP se crean los servicios y se exponen sus interfaces.



5 – Arranque del dispositivo DevPL1 (fig. 59):

```

osgi> 2021/06/26 15:20:25.509 TRACE Monitor : (Microservice-HealthStatus-monitor) Received measure: DevPL1 => H
2021/06/26 15:20:25.510 TRACE Monitor : Updating Knowledge Property DevPL1_HealthStatus with value DevPL1 => H
2021/06/26 15:20:25.513 TRACE LoopResource : <Rule microservice-healthstatus-productionline1-adaptationrule> Asking for the next System Configuration
2021/06/26 15:20:25.534 TRACE LoopResource : <Rule microservice-healthstatus-productionline1-adaptationrule> Request Planning (4d69eeae2fbb4e81aeae45f90bd15673)
2021/06/26 15:20:25.534 INFO AnalyzingModule : >>> ## STARTING NEW ADAPTATION PROCESS (4d69eeae2fbb4e81aeae45f90bd15673) <<<
2021/06/26 15:20:25.553 INFO AdaptiveReadyHost : [Host-DevPL1] DEPLOYING HOST
2021/06/26 15:20:25.554 INFO AdaptiveReadyHost : [Host-DevPL1] EXPOSING 10061 ENDPOINT
2021/06/26 15:20:25.555 INFO AdaptiveReadyHost : [Host-DevPL1] DEPLOYING DevPL1 CONTAINER
2021/06/26 15:20:25.556 INFO AdaptiveReadyContainer : [DevPL1] DEPLOYING DevPL1 CONTAINER IN Host-DevPL1 HOST
2021/06/26 15:20:25.557 INFO AdaptiveReadyHost : [kubernetes] DEPLOYING PL1 CONTAINER
2021/06/26 15:20:25.557 INFO AdaptiveReadyContainer : [PL1] DEPLOYING PL1 CONTAINER IN kubernetes HOST
2021/06/26 15:20:25.558 INFO AdaptiveReadyHost : [kubernetes] DEPLOYING TLC1 CONTAINER
2021/06/26 15:20:25.559 INFO AdaptiveReadyContainer : [TLC1] DEPLOYING TLC1 CONTAINER IN kubernetes HOST
2021/06/26 15:20:25.559 INFO AdaptiveReadyContainer : [DevPL1] DEPLOYING DevPL1 MICROSERVICE
2021/06/26 15:20:25.560 INFO AdaptiveReadyMicroservice : [DevPL1] DEPLOYING DevPL1 MICROSERVICE IN DevPL1 CONTAINER
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE DEPLOYMENT: pl1_deployment.embalpack
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE SERVICE: pl1-ars-svc.embalpack
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE DEPLOYMENT: pl1_deployment.embalpack (done)
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE SERVICE: pl1-ars-svc.embalpack (done)
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyContainer : [PL1] DEPLOYING PL1 MICROSERVICE
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroservice : [PL1] DEPLOYING PL1 MICROSERVICE IN PL1 CONTAINER
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE DEPLOYMENT: tlc1_deployment.embalpack
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE SERVICE: tlc1-ars-svc.embalpack
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE DEPLOYMENT: tlc1_deployment.embalpack (done)
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE SERVICE: tlc1-ars-svc.embalpack (done)
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyContainer : [TLC1] DEPLOYING TLC1 MICROSERVICE
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroservice : [TLC1] DEPLOYING TLC1 MICROSERVICE IN TLC1 CONTAINER
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroservice : [DevPL1] EXPOSING DevPL SERVICE THROUGH DevPL INTERFACE
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE SERVICE: pl1-pl-svc.embalpack
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroservice : [PL1] EXPOSING PL SERVICE THROUGH PL INTERFACE
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> CREATE SERVICE: tlc1-tlc-svc.embalpack
2021/06/26 15:20:25.560 DEBUG AdaptiveReadyMicroservice : [TLC1] EXPOSING TLC SERVICE THROUGH TLC INTERFACE
2021/06/26 15:21:03.211 INFO AdaptiveReadyMicroservice : [DevPL1] DEFINING DigitalTwins REQUIRED SERVICE THROUGH DigitalTwins INTERFACE
2021/06/26 15:21:03.211 INFO AdaptiveReadyMicroservice : [DevPL1] DEFINING ThingsREG REQUIRED SERVICE THROUGH ThingsReg INTERFACE
2021/06/26 15:21:03.212 INFO AdaptiveReadyMicroservice : [DevPL1] DEFINING Locations REQUIRED SERVICE THROUGH LOC INTERFACE
2021/06/26 15:21:03.212 INFO AdaptiveReadyMicroservice : [PL1] DEFINING DevPL REQUIRED SERVICE THROUGH DevPL INTERFACE
2021/06/26 15:21:03.212 INFO AdaptiveReadyMicroservice : [PL1] DEFINING PLWO REQUIRED SERVICE THROUGH PLWO INTERFACE
2021/06/26 15:21:03.213 INFO AdaptiveReadyMicroservice : [PL1] DEFINING Stock REQUIRED SERVICE THROUGH STOCK INTERFACE
2021/06/26 15:21:03.214 INFO AdaptiveReadyMicroservice : [PL1] DEFINING Transf REQUIRED SERVICE THROUGH TRANSF INTERFACE
2021/06/26 15:21:03.214 INFO AdaptiveReadyMicroservice : [TLC1] DEFINING DevPL REQUIRED SERVICE THROUGH DevPL INTERFACE
2021/06/26 15:21:03.214 INFO AdaptiveReadyMicroservice : [TLC1] DEFINING DevTL REQUIRED SERVICE THROUGH DevTL INTERFACE
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [TLC1] DEFINING DigitalTwins REQUIRED SERVICE THROUGH DigitalTwins INTERFACE
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [TLC1] DEFINING PLWO REQUIRED SERVICE THROUGH PLWO INTERFACE
2021/06/26 15:21:03.215 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> PRIVATE BINDING PL1:PLWO SERVICE TO http://erp-plwo-svc.embalpack:30002
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [PL1] PRIVATE BINDING PLWO TO PLWO INTERFACE OF ERP MICROSERVICE
2021/06/26 15:21:03.215 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> PUBLIC BINDING PL1:DevPL SERVICE TO http://pl1.embalpack.net:10061
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [PL1] PUBLIC BINDING DevPL TO 10061 ENDPOINT OF Host-DevPL1 HOST
2021/06/26 15:21:03.215 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> PRIVATE BINDING TLC1:PLWO SERVICE TO http://erp-plwo-svc.embalpack:30002
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [TLC1] PRIVATE BINDING PLWO TO PLWO INTERFACE OF ERP MICROSERVICE
2021/06/26 15:21:03.215 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> PUBLIC BINDING TLC1:DevPL SERVICE TO http://tlc1.embalpack.net:10061
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [TLC1] PUBLIC BINDING DevPL TO 10061 ENDPOINT OF Host-DevPL1 HOST
2021/06/26 15:21:03.215 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> PRIVATE BINDING ERP:TLC SERVICE TO http://tlc1-tlc-svc.embalpack:30029
2021/06/26 15:21:03.215 INFO AdaptiveReadyMicroservice : [ERP] PRIVATE BINDING TLC TO TLC INTERFACE OF TLC1 MICROSERVICE
2021/06/26 15:21:03.215 INFO ExecutorModule : Adaptation success. Updating the current configuration ...
2021/06/26 15:21:03.215 INFO AnalyzingModule : >>> ## ADAPTATION PROCESS ENDED (4d69eeae2fbb4e81aeae45f90bd15673) <<<
    
```

Figura 59. Observamos que en el monitor se recibe que DevPL1 marca ‘H’ como estado, lo que desencadena el proceso de adaptación. Se puede apreciar como se realiza el Deployment y la creación de servicios de PL1 y TLC1 en Kubernetes. Así mismo observamos las acciones Expose y Binding que se realizan sobre ARM-PL y ARM-TLC.

6 - Parada del dispositivo DevPL (figs. 60 y 61):

```

osgi> 2021/06/26 15:21:56.232 TRACE Monitor : (Microservice-HealthStatus-monitor) Received measure: DevPL1 => D
2021/06/26 15:21:56.233 TRACE Monitor : Updating Knowledge Property DevPL1_HealthStatus with value DevPL1 => D
2021/06/26 15:21:56.236 TRACE LoopResource : <Rule microservice-healthstatus-productionline1-adaptationrule> Asking for the next System Configuration
2021/06/26 15:21:56.246 TRACE LoopResource : <Rule microservice-healthstatus-productionline1-adaptationrule> Request Planning (6f14aa44091c477bb55232926f3f5bc7)
2021/06/26 15:21:56.246 INFO AnalyzingModule : >>> ## STARTING NEW ADAPTATION PROCESS (6f14aa44091c477bb55232926f3f5bc7) <<<
2021/06/26 15:21:56.275 INFO AdaptiveReadyMicroservice : [DevPL1] REMOVING SERVICE IN DevPL INTERFACE
2021/06/26 15:21:56.279 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> ELIMINAR SERVICE: pl1-pl-svc.embalpack
    
```

Figura 60. Vemos la recepción del nuevo estado del dispositivo – D – ahora MAPE-K realiza las operaciones del bucle para determinar el nuevo proceso de adaptación. Al haber cambio de estado, hay que modificar la arquitectura.

```

osgi> 2021/06/26 15:22:01.391 INFO AdaptiveReadyMicroservice : [PL1] REMOVING SERVICE IN PL INTERFACE
2021/06/26 15:22:01.391 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> ELIMINAR SERVICE: tlc1-tlc-svc.embalpack
2021/06/26 15:22:06.527 INFO AdaptiveReadyMicroservice : [TLC1] REMOVING SERVICE IN TLC INTERFACE
2021/06/26 15:22:06.529 INFO AdaptiveReadyContainer : [DevPL1] UNDEPLOYING DevPL1 MICROSERVICE
2021/06/26 15:22:06.529 INFO AdaptiveReadyMicroservice : [DevPL1] UNDEPLOYING MICROSERVICE
2021/06/26 15:22:06.534 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> ELIMINAR DEPLOYMENT: pl1_deployment.embalpack
2021/06/26 15:22:06.535 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> ELIMINAR SERVICE: pl1-ars-svc.embalpack
2021/06/26 15:22:11.995 INFO AdaptiveReadyContainer : [PL1] UNDEPLOYING PL1 MICROSERVICE
2021/06/26 15:22:11.995 INFO AdaptiveReadyMicroservice : [PL1] UNDEPLOYING MICROSERVICE
2021/06/26 15:22:11.998 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> ELIMINAR DEPLOYMENT: tlc1_deployment.embalpack
2021/06/26 15:22:11.998 DEBUG AdaptiveReadyMicroserviceEffectors4Kubernetes : ==>>> <api-server> ELIMINAR SERVICE: tlc1-ars-svc.embalpack
2021/06/26 15:22:17.142 INFO AdaptiveReadyContainer : [TLC1] UNDEPLOYING TLC1 MICROSERVICE
2021/06/26 15:22:17.142 INFO AdaptiveReadyMicroservice : [TLC1] UNDEPLOYING MICROSERVICE
2021/06/26 15:22:17.145 INFO AdaptiveReadyHost : [Host-DevPL1] UNDEPLOYING DevPL1 CONTAINER
2021/06/26 15:22:17.145 INFO AdaptiveReadyContainer : [DevPL1] UNDEPLOYING CONTAINER FROM Host-DevPL1 HOST
2021/06/26 15:22:17.146 INFO AdaptiveReadyHost : [kubernetes] UNDEPLOYING PL1 CONTAINER
2021/06/26 15:22:17.147 INFO AdaptiveReadyContainer : [PL1] UNDEPLOYING CONTAINER FROM kubernetes HOST
2021/06/26 15:22:17.147 INFO AdaptiveReadyHost : [kubernetes] UNDEPLOYING TLC1 CONTAINER
2021/06/26 15:22:17.147 INFO AdaptiveReadyContainer : [TLC1] UNDEPLOYING CONTAINER FROM kubernetes HOST
2021/06/26 15:22:17.165 INFO AdaptiveReadyHost : [Host-DevPL1] UNDEPLOYING HOST...
2021/06/26 15:22:17.165 INFO ExecutorModule : Adaptation success. Updating the current configuration ...
2021/06/26 15:22:17.191 INFO AnalyzingModule : >>> ## ADAPTATION PROCESS ENDED (6f14aa44091c477bb55232926f3f5bc7) <<<
    
```

Figura 61. Ya se ha determinado las operaciones a realizar para la nueva situación y el ejecutor envía órdenes al Efecto para llevar a cabo la adaptación.



M. Blanco Máñez

El resto de escenarios de adaptación se ejecutan correctamente, ofreciendo la salida que ofrece *FADA* los resultados esperados.

Con esto concluimos la parte de implementación y test, la siguiente fase es la implementación completa en la fábrica.





7. Conclusiones

En la introducción nos hacíamos la pregunta ¿y si pudiéramos crear un sistema auto-adaptativo basado en la arquitectura de microservicios utilizando *Kubernetes* como plataforma?

En este trabajo hemos diseñado e implementado un conjunto de servicios *adaptive ready* y se ha desplegado una arquitectura de microservicios reconfigurable sobre *Kubernetes* usando *FADA* como bucle de control. Aún hemos ido más allá y hemos utilizado un caso de estudio real en el que se aplican estos principios para desarrollar el sistema informático de la fábrica de producción industrial Embalpack.

Creo que podemos contestar afirmativamente a esta cuestión, dado que hemos conseguido extender las capacidades de *Kubernetes*, que a partir de ahora no sólo se podrá utilizar como una infraestructura que permite monitorizar y desplegar/replegar microservicios dinámicamente, sino además se podrá usar como una plataforma en la que crear arquitecturas dinámicamente.

Hemos aportado una mejora a una herramienta ya de por sí muy completa. Podemos disponer de un sistema auto-adaptativo, reconfigurando componentes y conexiones, con la garantía de disponibilidad, escalabilidad y rendimiento que ofrece *Kubernetes*.

En cuanto a la parte SAS – *Self Adaptive System* – hemos desarrollado e implementado varios componentes *adaptive-ready* que habían sido propuestos teóricamente en un anterior TFM [16], confirmando empíricamente que la propuesta es implementable.

Esto ha reportado diferentes beneficios, como desacoplar las políticas de adaptación de la implementación de los servicios, la posibilidad de reemplazar el bucle de control sin que afecte a la implementación de los microservicios, o de separar el desarrollo de los microservicios de la tecnología en la que se van a ejecutar y auto-adaptar.

Así mismo hemos obtenido la implementación de componentes reutilizables que ayudarán a desarrollar otras soluciones, como son los efectores del sistema para *Kubernetes* – *Efector* –, y la implementación de un instrumento de medir – *Sonda* – que podrá ejecutarse junto a los dispositivos sean cuales sean.

La tendencia en la industria ahora mismo está muy definida, es la revolución 4.0, donde el IoT es omnipresente y los servicios para dotar de infraestructura son vitales. Nuestra propuesta puede dar respuesta a muchos de los interrogantes que existen en la actualidad sobre auto-adaptación de procesos en producción. Sabemos que las investigaciones realizadas en sistemas auto-adaptativos se encaminan a mejorar los bucles de control, por lo que esta propuesta es bastante novedosa pues se centra en el componente y no en el bucle. La validación de la propuesta en una aplicación industrial real puede que haya abierto un camino a seguir.



El siguiente paso a dar es implementar la infraestructura completa, con lo que aumentará el número de servicios, sus conexiones y configuraciones. Esto haría que la toda la producción de la fábrica se adaptará a los distintos eventos que pudieran ocurrir.

7.1 Impacto del trabajo realizado

Una vez terminado el trabajo podemos afirmar que este ha impactado de dos maneras, una a nivel de investigación, contribuyendo técnicamente en el desarrollo de un artículo científico, y otra a nivel industrial, donde ha abierto el camino de aplicación en la fábrica Embalpack, sobre el que ya se está aplicando.

7.1.1 Colaboración con artículo científico

Simultáneamente al desarrollo de este trabajo se ha escrito un artículo científico que será presentado en las jornadas nacionales de ciencias y servicios – JCIS 2021 – que promueve la Sociedad de Ingeniería de Software y Tecnologías de Desarrollo de Software.

El artículo se denomina “*Servicios adaptive-ready para la reconfiguración dinámica de arquitecturas de microservicios*” [11] y ha sido confeccionado por el Centro de Investigación PROS de la Universitat Politècnica de València. En dicho artículo se describe exactamente nuestro escenario donde entran los ARS, *Kubernetes* y *FADA*.

En su punto 2 “Estado del arte” dice:

“El objetivo de este trabajo es proponer una estrategia que permita diseñar arquitecturas de microservicios que estén preparados para ser reconfigurados dinámicamente, extendiendo las capacidades de las infraestructuras y herramientas actuales. La propuesta debe introducir y diseñar componentes reutilizables, y debe poder aplicarse de manera amplia sobre este tipo de soluciones. Por último, debe ser concreta para poder implementarse con tecnologías actuales.”

Con nuestra aportación hemos dado validez práctica a la propuesta teórica del artículo, reafirmando y sentando las bases para posibles ampliaciones donde las arquitecturas sean más complejas y el número de servicios aumente. En un futuro próximo se podrían aplicar algoritmos de *machine-learning* de forma que el sistema fuera aprendiendo cómo suceden los eventos. En un momento dado de su aprendizaje podría adelantarse en el tiempo a dicho evento, aumentando considerablemente la eficiencia del proceso productivo.

Resaltar que en este documento se ha hecho un uso extenso de las referencias al artículo, tanto en texto como en figuras.

7.2 Aplicación industrial

Este trabajo también se puede situar como continuación de otro TFM “*Disseny i prototipat de solucions autoadaptatives emprant architectures basades en microserveis. Una aplicació industrial pràctica.*”[16], donde se abre un camino de investigación en el que conceptualiza la empresa como un conjunto de microservicios – y comienza a emplear contenerización –. En este TFM se identifican las limitaciones sobre *Kubernetes*.

En nuestro trabajo hemos dado el salto y ahora podemos reconfigurar dinámicamente la arquitectura de microservicios, extendiendo así las capacidades de *Kubernetes*. Creo que la combinación de estos dos trabajos ha abierto un camino a seguir por la industria, ya que proporciona capacidades de autogestión a un grupo de microservicios que se implementan sobre arquitecturas definidas, y que actúan directamente en el proceso productivo. Esto aumentará significativamente la eficiencia en la producción, disminuyendo los tiempos de respuesta ante eventos imprevistos, la máquina sabe qué ha pasado y cómo arreglarlo.

Como resumen decir que personalmente ha sido un privilegio poder participar en un proceso de investigación, donde a mi entender se revela el verdadero significado de ser ingeniero, innovar y crear soluciones utilizando herramientas existentes o construyéndolas.



8. Referencias

- [1] IBM Corporation, 2005. *An architectural blueprint for autonomic computing*. Whitepaper published by IBM Software Group.
- [2] Kephart, J. O. and Chess, D. M., 2003. *The vision of autonomic computing*. Published by the IEEE Computer Society. Ref. 0018-9162/03
- [3] Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., y Shaw, M., 2009. *Engineering Self-Adaptive Systems through Feedback Loops*. Artículo académico.
- [4] Joan Fons. Valencia, Marzo de 2020. *Especificación de sistemas auto-adaptativos*. DSIC – UPV
- [5] Arcaini, P., Riccobene, E., Scandurra, P., 2015. *Modeling and Analyzing MAPE-K Feedback Loops for Self-adaptation*. Artículo académico
- [6] Fowler, S.J., *Production Ready microservices*. Ed. O'Really, ISBN 978-1-491-96597-9, 2017
- [7] Richardson, C., 2018. *Microservices Patterns*. Ed. Manning. ISBN 9781617294549. Disponible en: <https://www.manning.com/books/microservices-patterns>
- [8] Stetson, C., *Microservices: Reference Architecture*, NGNIX, eBook, 2017
- [9] Joan Fons, Manoli Albert, Miriam Gil, Vicente Pelechano, 2017. *Developing Self-Adaptive Systems through adaptive-ready Components*. ProS Research Center, Department of Information Systems and Computation, Universitat Politècnica de València, Spain
- [10] Magnus Larsson, *Hands-On microservices with Spring Boot and Spring Cloud*. Published by Packt Publishers, September 2019. ISBN: 9781789613476
- [11] Joan Fons, Manoli Albert, Miriam Gil, Vicente Pelechano, 2021. *Servicios adaptive-ready para la reconfiguración dinámica de arquitecturas de microservicios*. Jornadas Nacionales de Ciencias y Servicios (JCIS), aceptada-Pendiente Publicación, 22-24 Septiembre, Málaga, 2021.
- [12] Fowler, M., *Microservices: a Definition of this New Architectural Term*. 2014. Disponible en: <https://martinfowler.com/articles/microservices.html>
- [13] Kreger, K., 2001. *Web Services Conceptual Architecture (WSCA 1.0)*. Published by IBM Software Group.
- [14] *RFC 5246 The Transport Layer Security (TLS) Protocol Version 1.2*, 2008. Request for comments refer to Internet standards track protocol. Disponible en: <https://tools.ietf.org/html/rfc5246>
- [15] *RFC 2818 HTTPS Over TLS*. Request for comments by The Internet Society (2000). Disponible en: <https://tools.ietf.org/html/rfc2818>

[16] Climent Penadés, J. (2020). *Disseny i prototipat de solucions autoadaptatives emprant architectures basades en microserveis. Una aplicació industrial pràctica*. Disponible en: <http://hdl.handle.net/10251/153180>

En este anexo comentaremos aquellas cuestiones o problemas que han sido de especial relevancia, como son dar acceso a *Minikube* a nuestro repositorio privado. Comentaremos también los distintos “camino” – todos útiles – a los que nos ha llevado este proyecto, por ejemplo el uso de *Prometheus* y *Alertmanager* alojados en *Kubernetes* como *Sondas* en nuestro escenario o la creación de un clúster *Kubernetes* utilizando tres máquinas *Ubuntu*.

1. Cómo accede *Minikube* al repositorio privado.

Para realizar una petición de descarga de imagen en *Docker registry* se utiliza REST, esto significa que son peticiones basadas en HTTP. Este protocolo ha ido evolucionando hacia una manera más segura de comunicarse con la implantación de otra capa de protocolo llamada TLS [12]– *Transport Layer Security* –, naciendo así HTTPS [13]. Este protocolo cifra los mensajes en el cliente y los descifra en el servidor, o viceversa, consiguiendo así una comunicación segura. Esto lo hace basándose en certificados.

Bien, pues para acceder a *Docker registry* primero hay una autenticación, que demanda usuario y contraseña y una segunda fase es la negociación TLS donde se comprueban los certificados que poseemos. Si todo es correcto podemos descargarnos la imagen.

Existen dos tipos de certificados, el autorizado por una Agencia Certificadora que da fe de que el certificado pertenece a quien dice ser, y otro auto-firmado que no cuenta con ninguna garantía de autenticidad.

Así pues *Minikube/Kubernetes*, no admite en su política por defecto certificados auto-firmados, y como no podía ser de otra forma hace la comprobación y muestra un error en la descarga de la imagen. Como ya se ha dicho no disponemos de un certificado de esa categoría con lo que se tiene que realizar una configuración extra para que *Minikube* trabaje con nuestro *Docker registry*.

En proyectos que utilizan tecnología que está en el mercado la forma más habitual de conseguir información es la investigación por internet, consultando documentación oficial de la herramienta, webs o en foros de desarrolladores. En este tipo de investigación cuando se junta que la documentación oficial no contempla ese caso, que la búsqueda es de una herramienta que no la utiliza la comunidad en general y que la consulta es muy específica este proceso puede tornarse muy frustrante. Pero con perseverancia y paciencia al final se consigue.

En *Kubernetes* existe un objeto llamado *DaemonSet* (ver 3.4.1) que realiza acciones en segundo plano cada vez que se crea un *Pod* en el clúster. En el archivo de creación de *DaemonSet* se pueden definir acciones sobre el sistema, como es la copia de archivos desde la máquina anfitrión a *Kubernetes*.

Para que *Kubernetes* utilice nuestro certificado auto-firmado hay que guardarlo en la carpeta `/etc/docker/certs.d/` en el core del mismo. Así pues podemos automatizar esta acción utilizando *DaemonSet* para que cada vez que se haya de crear un *Pod* nos aseguremos que el certificado se encuentra en su sitio (*fig. 1*).

```
spec:
  containers:
  - name: registry-ca
    image: busybox
    command: [ 'sh' ]
    args: [ '-c', 'cp /home/core/registry-ca /etc/docker/certs.d/tambori.dsic.upv.es:10050/ca.crt && exec tail -f /dev/null' ]
    volumeMounts:
    - name: etc-docker
      mountPath: /etc/docker/certs.d/tambori.dsic.upv.es:10050
    - name: ca-cert
      mountPath: /home/core
    terminationGracePeriodSeconds: 30
  volumes:
  - name: etc-docker
    hostPath:
      path: /etc/docker/certs.d/tambori.dsic.upv.es:10050
  - name: ca-cert
    secret:
      secretName: registry-ca
```

Figura 1. Fracción de código del archivo de creación, donde resaltamos el nombre de nuestro repositorio.

Hasta aquí todo bien, pero como ya se ha dicho, en aras de un mejor control de los recursos, *Kubernetes* es muy restrictivo a la hora de nombrar objetos y archivos. Así pues, como el nombre de nuestro repositorio contiene el carácter “ : ” para definir el puerto, eso contraviene la política de *Kubernetes* por lo que el objeto *DaemonSet* no se crea.

Una solución era cambiar el nombre del repositorio, pero no a lugar por ser un repositorio compartido, había que mantener el nombre. Al final encontramos una forma de realizarlo, aunque hay que decir que es poco eficiente, funciona correctamente.

La solución es entonces:

- Copiar el contenido del certificado auto-firmado `certificado.crt` que tenemos en `/etc/docker/certs.d/tambori.dsic.upv.es:10050` de nuestra máquina que es con el que accedemos y descargamos manualmente.
- Acceder por `ssh` a *Minikube*.
- Crear la carpeta `/etc/docker/certs.d/` en *Minikube*.
- Dentro de `certs.d` crear la carpeta de nuestro registro → `tambori.dsic.upv.es:10050`
- Dentro de `tambori.dsic.upv.es:10050` crear el archivo `certificado.crt` y pegar el contenido que anteriormente habíamos copiado.

Con esto *Minikube* reconoce el certificado y permite la descarga desde nuestro *Docker registry*.

2. Prometheus y Alertmanager

Prometheus es un conjunto de herramientas de monitorización y alerta de sistemas de código abierto construido originalmente en *SoundCloud*. Es un software con soporte nativo para *Kubernetes*. Monitorizar los clústeres de *Kubernetes* con *Prometheus* es una opción natural porque muchos componentes de *Kubernetes* incluyen métricas en formato *Prometheus* de forma predeterminada y, por lo tanto, las puede descubrir fácilmente.

En la fase de diseño del trabajo se consideró el uso de *Prometheus* como *Sonda* para monitorizar los microservicios en *Kubernetes*, se implementó entonces para testear su funcionalidad y decidir si incluirlo o no en nuestra solución. Cuando se decidió el escenario de adaptación, vimos que *Prometheus* en *Kubernetes* si bien es una herramienta muy potente no nos servía para el propósito del escenario, ya que los dispositivos que se miden están fuera del clúster. En cualquier caso está configurado y funcionando en *Kubernetes*, ya que, si en un momento dado se decide monitorizar los microservicios la herramienta está lista para usarse.

Para instalar *Prometheus* en *Kubernetes* hay que utilizar estos objetos:

- *Deployment*
- *Service*
- *ServiceAccount*
- *ClusterRole*
- *ClusterBinding*
- *ConfigMap*

ClusterRole, *ClusterBinding*, *ServiceAccount* → si queremos recuperar datos de todo el clúster para dar acceso a *Prometheus* a todos los recursos del mismo hay que configurar *Kubernetes* para conseguirlo.

ConfigMap → archivo de configuración de *Prometheus*. Aquí establecemos que objetos tiene que medir – *Pods*, *Services*, ... – también estipulamos el evento que debe lanzar una alerta. Gracias a *PromQL*, un lenguaje de consulta flexible se pueden definir multitud de alertas, desde paradas de funcionamiento, a consumo de CPU pasando por el número de consultas a un determinado servicio. Así mismo se puede establecer el intervalo en que se realizan las medidas y determinar a quién se va a enviar la alerta generada (*fig. 2*).



```

data:
  prometheus.rules: |-
    groups:
    - name: alert
      rules:
      - alert: InstanceDown
        expr: up{app="erp-ars-dpl"} == 0
        for: 5s
        labels:
          severity: ticket
  prometheus.yml: |-
    global:|
      scrape_interval: 5s
      evaluation_interval: 5s
    rule_files:
    - /etc/prometheus/prometheus.rules
    alerting:
      alertmanagers:
      - scheme: http
        static_configs:
        - targets:
          - "alertmanager-service.monitoring:9093"
    
```

Figura 2. Fracción del archivo de configuración de Prometheus.

Interfaz gráfica Prometheus:

- Pods monitorizados:

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.13:8080/actuator/prometheus	UP	app="erp" instance="172.17.0.13:8080" job="kubernetes-pods" kubernetes_namespace="embalpack" kubernetes_pod_name="erp-dpl-68cb845b5d-8t28c" pod_template_hash="68cb845b5d" type="DevOps"	35.932s ago	9.846ms	
http://172.17.0.5:8080/actuator/prometheus	UP	app="rosa" instance="172.17.0.5:8080" job="kubernetes-pods" kubernetes_namespace="test" kubernetes_pod_name="rosa-deployment-96bc9c694-j4vik" pod_template_hash="96bc9c694" type="DevOps"	36.222s ago	8.953ms	

- Secuencia en el tratamiento de una alerta:

➤ A la espera.

Inactive (1)
 Pending (0)
 Firing (0)

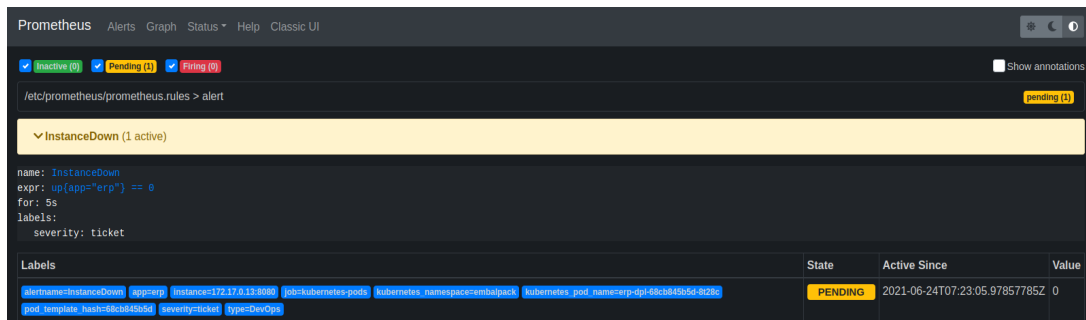
/etc/prometheus/prometheus.rules > alert

▼ InstanceDown (0 active)

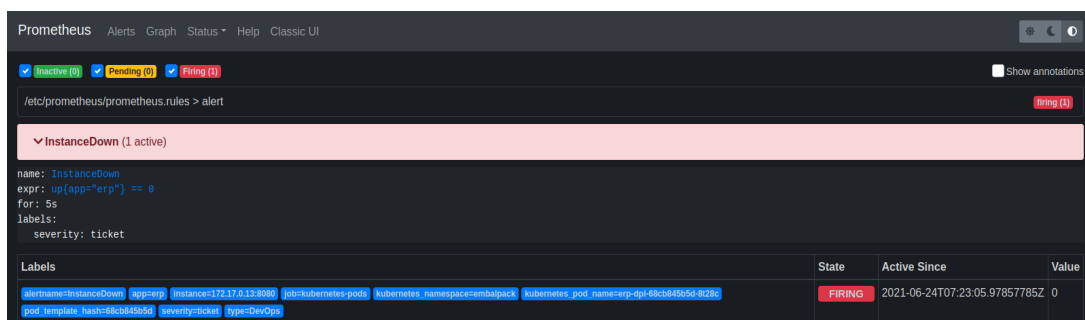
```

name: InstanceDown
expr: up{app="erp"} == 0
for: 5s
labels:
  severity: ticket
    
```

- Pendiente de lanzar. *Prometheus* espera un intervalo de tiempo para lanzar la alerta, el *Pod* se podría recuperar del fallo y no sería necesario el envío. Este tiempo es configurable.



- Lanzando la alerta. Una vez superado el tiempo establecido se lanza la alerta, que se enviará a *Alertmanager*.



Alertmanager es un sistema de alerta de código abierto que funciona con el sistema de monitorización *Prometheus*. *Alertmanager* maneja las alertas enviadas por aplicaciones cliente como el servidor *Prometheus*. Se encarga de duplicarlas, agruparlas y enrutarlas al receptor que definamos, como correo electrónico, o una URL que dirija la alerta a nuestro monitor. También se encarga de silenciar e inhibir alertas.

Como *Alertmanager* va de la mano de *Prometheus* cuando se prescindió de este, *Alertmanager* tampoco era necesario. De igual manera está instalado y configurado para que envíe las alertas al monitor, por si en el futuro se desea utilizar.

Para instalar *Alertmanager* en *Kubernetes* hay que utilizar estos objetos:

- *Deployment*
- *Service*
- *ConfigMap*

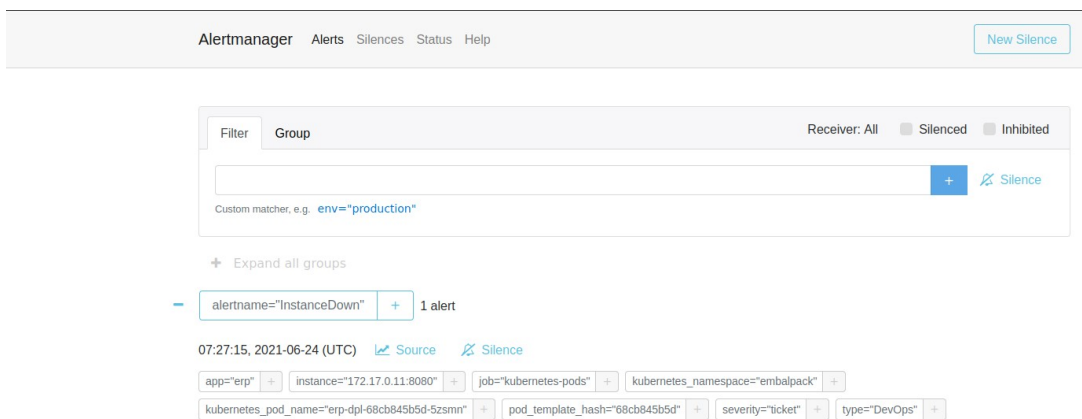


ConfigMap → En este archivo configuramos a que grupo pertenece la alerta, el receptor y la URL destino del mensaje de alerta (*fig. 3*).

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: alertmanager-config
  namespace: monitoring
data:
  config.yml: |-
    global:
      resolve_timeout: 1m
    route:
      receiver: log-alerts
      group_by: ['alertname']
      group_wait: 5s
      repeat_interval: 1m
      routes:
        - match:
            severity: ticket
            group_wait: 5s
            repeat_interval: 20s
            receiver: log-alerts
    receivers:
      - name: log-alerts
        webhook_configs:
          - url: http://192.168.1.90:7092/fada/monitor
```

Figura 3. Archivo de configuración de Alertmanager.

Vemos la interfaz gráfica de *Alertmanager* en el momento de recibir el aviso desde *Prometheus*.



3. Creación de un clúster *Kubernetes*

Para la realización de este trabajo hemos elegido *Minikube* por ser la solución ideal para pequeños proyectos basados en contenedores. Permite, por ejemplo, configurar un clúster de *Kubernetes* en privado sin tener que trabajar directamente con todo un servidor o una nube. Pero somos conscientes que cuando se trata de empresas van a trabajar con servidores o haciendo *cloud-computing*, por lo que también exploramos esta solución.

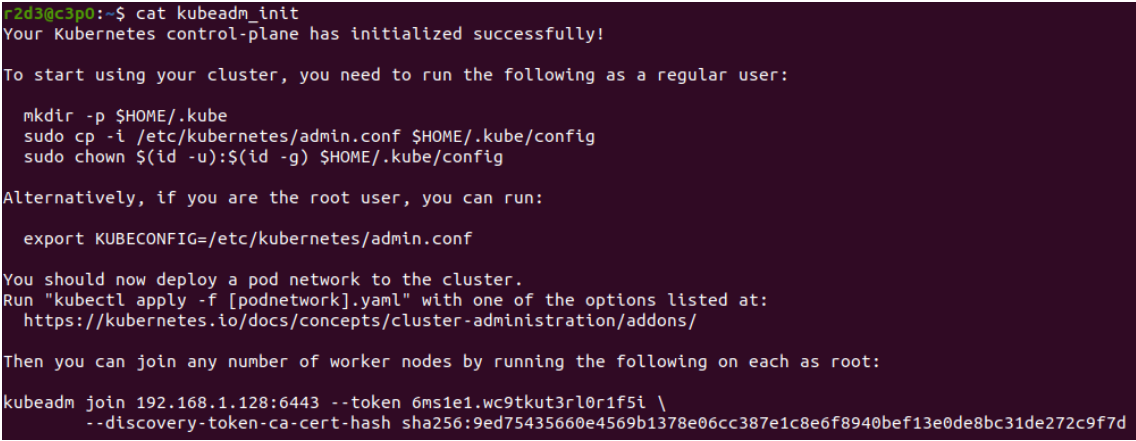
Así pues creamos un clúster con tres máquinas *Ubuntu*. Para el funcionamiento de *Kubernetes* hay que declarar un nodo *master* y tantos *workers* como se crea conveniente. Este nodo *master* será donde se realicen todas las operaciones y acciones, que después serán exportadas a los *workers*, esto se hace con `kubectl`. *Kubernetes* se encargará de alojar los *Pods* de una manera transparente al usuario, es una de sus virtudes. Después gracias a *Service* se podrá acceder a los *Pods* – da igual donde estén alojados – utilizando una sola URL.

Se deben instalar `kubeadm`, `kubelet`, y `kubectl` en todos los nodos

El paquete en el que se diferencia de *Minikube* es `kubeadm`. Este paquete sólo se inicia en el nodo *master* y sirve para dotarle de infraestructura al clúster permitiendo la propagación de configuraciones o creaciones en *master*.

Cuando el *master* ya esta configurado se hace un `init` en `kubeadm`. A continuación sale por consola un mensaje informativo (*fig. 4*).

```
sudo kubeadm init --pod-network-cidr=192.168.0.0/16
```



```
r2d3@c3p0:~$ cat kubeadm_init
Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 192.168.1.128:6443 --token 6ms1e1.wc9tkut3rl0r1f5i \
  --discovery-token-ca-cert-hash sha256:9ed75435660e4569b1378e06cc387e1c8e6f8940bef13e0de8bc31de272c9f7d
```

Figura 4. Mensaje informativo que ofrece `kubeadm` para completar la implementación del clúster. Es importante el último comando, ya que es el que enlaza *master* y *workers* utilizando *tokens*.

Ahora simplemente hay que seguir las instrucciones y realizarla en todos los *workers*.



Después de realizar varias pruebas, tenemos que decir que la elección de *Minikube* ha sido acertada, gracias a su simplicidad y sobre todo a la interfaz gráfica hemos aprendido más rápidamente y de una manera intuitiva el funcionamiento de *Kubernetes*. Desde aquí recomendamos su uso para ‘romper la mano’ en el uso de plataformas que trabajan en la nube.

Con esto terminamos este anexo en el que hemos incluido algunas cosas que queríamos que aparecieran en este documento.

