UNIVERSIDAD
DE GRANADA

Escuela Técnica Superior en Ingenierías Informática y de Telecomunicación

Máster Universitario en Ingeniería Informática

TRABAJO DE FIN DE MÁSTER

Sistema de Fog-Computing para Control Distribuido

Presentado por:
Víctor Vázquez Rodríguez

Tutor:
Antonio Javier Díaz Alonso
Departamento de Arquitectura y Tecnología de Computadores

Curso académico 2020-2021

Agradecimientos

Al Dr. Antonio Javier Díaz Alonso, tutor de este trabajo, por guiarme y aportarme su experiencia académica.

A mis padres, por darme la oportunidad de estudiar y seguir mi propio camino.

Al Dr. Juan Antonio Vázquez Rodríguez, mi hermano, por ser mi referente de esfuerzo, trabajo, bondad y humildad.

A mi novia, Rosa, por creer siempre en mí y estar a mi lado pase lo que pase.

A mis compañeros de promoción del Máster, por hacerme sentir como en casa lejos de ella.

Resumen

Con el advenimiento de la Industria 4.0, las empresas buscan incorporar técnicas de inteligencia artificial y análisis de datos a sus instalaciones y procesos industriales con el objetivo de mejorar la productividad y la autonomía. En este trabajo, se plantea la posibilidad de usar tecnologías de contenerización de procesos para el despliegue eficiente de estas nuevas tareas junto con las de tiempo real habituales en los sistemas de control industrial, estudiando las distintas tecnologías posibles y realizando un análisis del rendimiento de tareas de tiempo real contenerizadas con Docker. Además, se diseña e implementa una herramienta software que sirve como prueba de concepto para el despliegue y la orquestación de este tipo de tareas sobre entornos distribuidos mediante el uso de contenedores.

As Industry 4.0 gets closer, companies are looking to incorporate artificial intelligence and data analysis techniques to their facilities and industrial processes with the objective of improving productivity and autonomy. This paper proposes the use of processes containerization techonologies for efficient deployment of these new tasks along with the real-time tasks that are common in industrial control systems, studying different possible techonologies and analysing the performance of real-time tasks containerized using Docker. Finally, a proof-of-concept software tool for deployment and orchestration of this type of tasks on distributed environments by means of containers.

Índice general

Índice de figuras

Índice de tablas

Índice de listados de código

1. Introducción

1.1. Motivación y contexto

El Programma 101 es considerado por muchos como el primer ordenador de uso personal. Producido por la empresa italiana Olivetti entre los años 1962 y 1964, este dispositivos seasimilaba más a una calculadora que al concepto de ordenador que se tiene en la actualidad. Casi 60 años después, los ordenadores han evolucionado hasta convertirse en un objeto asequible y casi indispensable, teniendo la mayor parte de la sociedad acceso a algún dispositivo con capacidad de computación. Aspectos de nuestra vida como el ocio, serían muy diferentes sin las redes sociales o los servicios online. Esta evolución de los ordenadores y la computación en general continúa hoy en día, buscando conectar dispositivos cotidianos a internet e incorporar en ellos cierta capacidad de procesamiento. Televisores inteligentes en los que instalar aplicaciones, robots aspiradores que funcionan de manera autónoma, frigoríficos que permiten ver el estado de los alimentos que almacenan, todos estos productos surgen de la expansión de los ordenadores hasta todos los rincones de nuestras vidas, con el objetivo final de hacerla más sencilla para los humanos. Esta expansión no está ocurriendo solo en los hogares, si no que se avanza hacia un mundo más conectado donde aparentemente todo lo que nos rodea tenga acceso a la red.

La industria no es ajena a este cambio, ya que la aplicación del internet de las cosas (IoT por sus siglas en inglés) a los procesos industriales podría aportar beneficios muy importantes como son el aumento de la productividad, de la calidad de los productos o de la seguridad de las instalaciones. No obstante, se trata de un proceso de implantación complejo y de muy largo plazo, debido a los estrictos requisitos de algunos sectores industriales. Las fábricas y plantas de producción relegan las tareas de control de sus operaciones a los sistemas de control industriales o ICSs (*Industrial Control Systems*). Normalmente, estos sistemas deben responder ante eventos que ocurren en las instalaciones en ventanas de tiempo muy pequeñas, dependiendo del proceso concreto y sus riesgos asociados. La fiabilidad de estos sistemas y su tolerancia ante los fallos es de gran importancia, siendo crucial la verificación de estos aspectos durante su diseño y desarrollo. Es por estos requisitos tan estrictos que, en muchos casos, para la implantación de los ICSs se utilizan tanto *hardware* como *software* específicos y que ya han sido validados para este tipo de aplicaciones. No obstante, el uso de estas herramientas también plantea algunos inconvenientes, como por ejemplo la difícil interoperabilidad con otros sistemas debido a la naturaleza cerrada de las mismas o la reutilización del código en otras plataformas, lo que acorta la vida útil del *software*.

Desde esta situación se afronta el avance hacia la cuarta revolución industrial o Industria 4.0. Uno de los objetivos principales que se persigue es la automatización completa de los procesos industriales, haciando que sean independientes y autogestionados, aumentando así su eficiencia, productividad y seguridad. Para conseguir este objetivo, es necesario incorporar a estos procesos las nuevas técnicas de aprendizaje automático y análisis de grandes volúmenes de datos, construyendo sistemas experots capaces de tomar decisiones propias para su funcio-

namiento y gestión (p. ej., modificar el ritmo de producción en base a predicciones sobre la demanda). Aunque estos sistemas expertos se podrían desplegar en plataformas separadas de las que habitan los ICSs, esto duplica los costes de infraestructura y hace también más difícil el mantenimiento de la misma. Por ello, hay una tendencia cada vez mayor hacia la ejecución de tareas con distintos niveles de criticalidad en una misma plataforma. Por otra parte, las tareas de aprendizaje máquina suelen requerir de una capacidad de computación relativamente elevada, sobre todo cuando la cantidad de datos sobre la que se trabaja es grande. Un solo dispositivo no sería capaz de gestionar estas tareas de forma eficiente si, además, debe realizar tareas críticas de control, dificultando también que pueda cumplir con sus requisitos en ese aspecto. Aparecen entonces nuevos paradigmas de computación, como el *fog* o el *edge computing*, que pretenden decentralizar el procesamiento, alejándolo de la nube y llevándolo a elementos intermedios más cercanos a las fuentes de datos o a los dispositivos del borde de la red, respectivamente. En estos paradigmas, se persigue la máxima utilización de la capacidad de computación de los dispositivos presentes en la red, distribuyendo entre ellos las tareas necesarias de forma dinámica.

Dentro de este campo, se están llevando a cabo muchas investigaciones para validar la efectividad de estos nuevos modelos y hacerlos realidad, además de comprobar su efectividad en el ámbito industrial. Uno de los planteamientos más prometedores es el que conlleva la aplicación de tecnologías de virtualización a los sistemas de control industrial, permitiendo su despliegue y ejecución distribuida en convivencia con otros procesos. Estas tecnologías, que han sufrido un desarrollo masico en los últimos años debido al auge de la computación en la nube, pueden ahora ser claves para asegurar la resiliencia y robustez de los ICSs en entornos distribuidos.

Por todo esto, este trabajo abordará la complejidad de los problemas descritos intentando buscar mecanismos que simplifiquen la gestión de los sistemas *fog*/*edge*. Partiremos de la experiencia previa en trabajos con contenedores orientados al análisis de sus prestaciones para la ejecución de tareas con restricciones temporales, responsables de las ideas aquí propuestas y base fundamental de parte de la motivación de este trabajo. Además, esta línea de investigación tiene mucho potencial para mejorar los sistemas de control implementados en la actualidad, permitiendo por tanto una contribución importante tanto en aspectos industriales como científicos, lo que aumenta aún más nuestro interés en esta temática.

1.2. Objetivos

De cara a la realización de este trabajo, se han planteado los siguientes objetivos a cumplir:

- Comprender los conceptos básicos sobre sistemas operativos de tiempo real, algoritmos de planificación y sistemas de control industriales.
- Analizar las soluciones de tiempo real basadas en GNU/Linux existentes.
- Estudiar la viabilidad y rendimiento de las tecnologías de contenerización para la ejecución de tareas con restricciones temporales.
- Diseñar e implementar una herramienta de despliegue de tareas de tiempo real mediante contenedores que sirva como prueba de concepto.

- Caracterizar las prestaciones de la herramienta desarrollada, identificando posibles aplicaciones y/o mejoras de la misma.

Una parte relevante de los objetivos del presente proyecto están asociados al estudio y caracterización de tecnologías, así como al análisis del mercado, que se justifica por un enfoque innovador orientado al desarrollo de sistemas para *fog computing*. Esto, junto con los conocimientos adquiridos sobre implementación de sistemas de tiempo real, será clave para que la herramienta desarrollada como prueba de concepto sea lo más eficiente y funcional posible.

Por otra parte, en los objetivos también se hace hincapié en el uso de soluciones basadas en GNU/Linux, como parte de un fuerte compromiso con las herramientas libres y de código abierto. Algunos de los sistemas operativos de tiempo real más conocidos y usados son de código propietario y es necesario adquirir licencias para utilizarlos, lo cual no favorece ni el aprendizaje ni la reutilización del software desarrollado para estas plataformas. Últimamente, se han realizado contribuciones importantes al kernel de Linux para mejorar el soporte que ofrece a cargas de trabajo de este tipo, además de los múltiples proyectos más especializados que está llevando a cabo la comunidad. En esta aproximación, hemos considerado primordial la extensión del software libre, apoyando su uso siempre que sea posible. Por ello, la herramienta planteada se diseñará en torno al despliegue de tareas en sistemas GNU/Linux.

1.3. Planificación

Tradicionalmente, la planificación de las tareas en proyectos de desarrollo de software se realizaba siguiendo un modelo en cascada, donde antes de avanzar a la siguiente fase (p. ej., diseño, desarrollo, prueba) se debían completar todas las tareas de la anterior. Además, se intentaba estimar con precisión el tiempo necesario para desarrollar cada tarea, con el fin de obtener una predicción precisa de cúando estaría terminado el proyecto. Este tipo de metodología en cascada, en la que se planifica la totalidad del proyecto de antemano, es muy rígida y, por tanto, hace que sea más difícil adaptarse a los cambios que suceden durante el desarrollo del proyecto. En los últimos años ha cobrado fuerza una alternativa: las metodlogías ágiles. Lo que se plantea es afrontar la planificación y ejecución de las tareas de forma iterativa. En SCRUM, una de las metodologías ágiles más conocidas, se eligen tareas a realizar a corto plazo, lo que permite modificar las tareas a realizar si fuera necesario. En [1] se realiza una comparativa entre el modelo en cascada y las metodologías ágiles, llegando el autor a la conclusión de que estas últimas son siempre más eficaces para proyectos de pequeño y medio tamaño si se aplican de forma correcta. Es en los proyectos más complejos, donde hay múltiples equipos involucrados, en los que la aplicación de metodologías ágiles es más difícil y puede resultar ineficaz.

Observando esto y sabiendo que este proyecto es de baja complejidad, hemos decidido aplicar SCRUM para la planificación y la gestión de las tareas del mismo. Según este modelo, el desarrollo del proyecto se realiza en iteraciones, las cuales se planifican de forma dinámica y sobre la marcha. Al inicio del proyecto, se identifican las tareas que se consideran necesarias para la consecución de los objetivos, formando una pila o *backlog* de tareas. Los miembros del equipo estiman estas tareas en base a su experiencia. A diferencia del modelo tradicional, estas estimaciones no tienen que ser precisas, sino que lo que se busca es una medida orientativa del tamaño o complejidad de las tareas en relación con las demás. En muchos casos, para agilizar

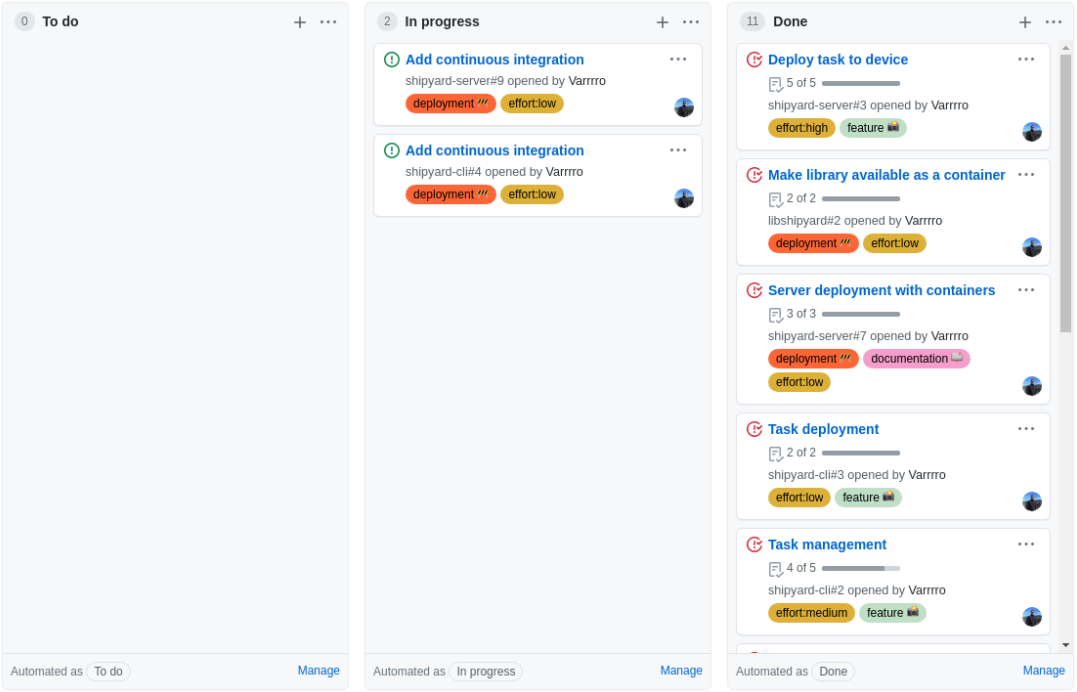


Figura 1.1.: Muestra del tablero Kanban usado para el seguimiento de las tareas del proyecto.

el proceso de estimación, se aplican juegos. Una vez definida esta pila de tareas, comienzan las iteraciones o *sprints*, que suelen tener una duración pequeña, de 2 o 3 semanas. Antes de comenzar una nueva iteración, los miembros del equipo deciden qué tareas se van a acometer en la misma. La selección de estas tareas se puede basar en su estimación, su importancia o en el propio deseo de los miembros del equipo de realizarlas. Así, se realiza una planificación a corto plazo, lo que permite a los equipos tener mucho margen de reacción ante cambios en el proyecto (p. ej., cambios en los requisitos o problemas imprevistos). Una vez termina la iteración, el equipo evalúa el trabajo realizado, comprobando si las estimaciones que habían realizado al principio eran fidedignas, actualizando las de las tareas restantes si fuera necesario. Si se ha descubierto que se deben realizar más tareas de las inicialmente planteadas, éstas se pueden añadir a la pila entre iteraciones.

Como se puede apreciar, el modelo de SCRUM se centra mucho en el trabajo colaborativo dentro del equipo, por lo que el modelo que hemos aplicado en este proyecto se puede considerar como un SCRUM adaptado al trabajo individual. Las iteraciones seguidas han sido de dos semanas y se ha usado un tablero Kanban para el seguimiento de las tareas. En la figura 1.1 se muestra el tablero usado. El objetivo de estos tableros es ofrecer una visión clara del estado del proyecto con un simple vistazo, para lo que ordenan las tareas en tres columnas: Por hacer (*To do*), Haciendo (*In progress*) y Hecho (*Done*). A continuación, se listan las iteraciones realizadas con las tareas que se han acometido en cada una de ellas:

- **Iteración 1** (06/07 - 20/07)
  - (Documentación) Estudiar el funcionamiento de los RTOS.

  - (Documentación) Estudiar las soluciones de tiempo real para Linux.
  - (Documentación) Estudiar las técnicas de diseño e implementación de tareas de control.
  - (Documentación) Estudiar el uso de contenedores para tareas de tiempo real.

- **Iteración 2** (20/07 - 03/08)
  - (Servidor) Implementar la obtención, inserción y eliminación de nodos.
  - (Servidor) Implementar la obtención, inserción y eliminación de tareas.

- **Iteración 3** (03/08 - 17/08)
  - (Servidor) Implementar el despliegue y la eliminación de tareas en los nodos.

- **Iteración 4** (17/08 - 31/08)
  - (Servidor) Implementar la obtención, inserción y eliminación de tareas.
  - (Servidor) Implementar el despliegue y la eliminación de tareas en los nodos.
  - (Servidor) Implementar la actualización de nodos y tareas.
  - (Librería) Implementar las funciones de manipulación de los atributos de planificación.
  - (Librería) Crear imagen base.

- **Iteración 5** (31/08 - 14/09)
  - (Servidor) Mejorar el manejo de los errores.
  - (Servidor) Definir despliegue mediante contenedores.
  - (Cliente) Implementar la gestión de los nodos.
  - (Cliente) Implementar la gestión de las tareas.
  - (Cliente) Implementar el despliegue de tareas.

- **Iteración 6** (14/09 - 28/09)
  - (Servidor) Implementar el despliegue y la eliminación de tareas en los nodos.
  - (Cliente) Implementar la gestión de los nodos.
  - (Cliente) Implementar el despliegue de tareas.

  - (Librería) Implementar las funciones de manipulación de los atributos de planificación.
  - (Librería) Crear imagen base.

- **Iteración 7** (28/09 - 12/10)
  - (Experimentación) Caracterizar el impacto de los contenedores sobre el rendimiento en tareas de tiempo real.

- **Iteración 8** (12/10 - 26/10)
  - (Experimentación) Caracterizar el impacto de los contenedores sobre el rendimiento en tareas de tiempo real.

- **Iteración 9** (26/10 - 09/11)
  - (Experimentación) Caracterizar el rendimiento del sistema desarrollado.

- **Iteración 10** (09/11 - 23/11)
  - (Servidor) Definir flujo de integración continua.
  - (Cliente) Definir flujo de integración continua.
  - (Librería) Definir flujo de integración continua.

Como se puede apreciar, las tareas se han diferenciado según su tipo o el componente del sistema desarrollado al que se refieren. Además, para algunas de estas tareas ha sido necesario dedicar más de una iteración. En el caso de la tarea de implementación de las operaciones de obtención, inserción y eliminación de tareas, se tuvo que volver a incorporar en la iteración 4 debido a algunos errores cometidos en la implementación y que se detectaron más tarde. Otro caso similar fue la implementación de la funcionalidad de despliegue de tareas en los nodos. En el código inicialmente escrito en las iteraciones 3 y 4, esta operación se llevaba a cabo abriendo directamente una conexión SSH con el nodo objetivo y ejecutando los comandos de despliegue, emulando lo que haría un operario humano. Posteriormente, se llegó a la conclusión de que esta implementación se podía mejorar la usar la librería oficial de Docker para Python, con la cuál se podía establecer una conexión al motor Docker del nodo remoto para realizar el despliegue. Estas situaciones, junto con otras dadas en más tareas, reflejan fielmente la capacidad de adaptarse a los imprevistos que ofrece SCRUM que ya hemos comentado anteriormente.

1.4. Material y métodos

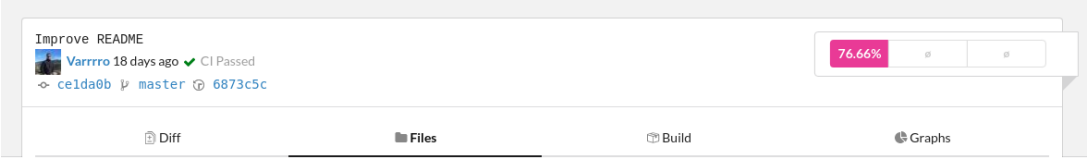
En esta sección, se presentan todas las herramientas que se han usado para llevar a cabo este proyecto, además de las metodologías de trabajo seguidas. En la figura 1.2 se puede ver una recopilación de los logotipos de todas estas herramientas, las cuales se van a exponer en detalle a continuación.

En un proyecto de desarrollo de código como es este, una de las herramientas más importantes es la de control de versiones. En nuestro caso, se ha usado **Git** debido a que es una herramienta de código libre y, además, es la más conocida y usada. El repositorio remoto se ha alojado en **GitHub**, la popular plataforma de desarrollo de código colaborativo. El desarrollo de código como tal se ha hecho con **Visual Studio Code**, un editor de texto de Microsoft que, aunque ofrece menos funcionalidad de serie que los entornos de desarrollo (IDEs) especializados, también es mucho más liviano y rápido. Además, esta funcionalidad se puede extender enormemente mediante extensiones, pudiendo convertirlo casi en un IDE adaptado a las necesidades concretas del usuario. Gracias a estas extensiones, se puede usar el mismo editor para distintos programar en distintos lenguajes, lo cual ha sido una de las principales razones por las que se ha escogido.

Figura 1.2.: Logotipos de las herramientas utilizadas. De izquierda a derecha yde arriba a abajo: Python, Docker, Raspberry Pi, GitHub, VS Code y draw.io.

Como ya se ha indicado en la sección anterior, se ha aplicado un SCRUM adaptado como metodología de planificación, gestión y seguimiento del proyecto, usando un tablero Kanban para la visualización de las tareas. Concretamente, se ha usado el tablero que ofrece GitHub como parte de sus herramientas para la gestión de proyectos. Las tareas se añaden a los repositorios como *issues*, los cuales se añaden al tablero para controlar su estado de realización. El beneficio que nos aporta el uso de GitHub para esto es el de tener una plataforma unificada tanto para la gestión del proyecto como para su desarrollo. Esta integración permite flujos de trabajo muy interesantes, como por ejemplo referenciar los *issues* ya mencionados en los mensajes de *commit*, de forma que se vinculan los cambios realizados con la tarea en la que se engloban. La gestión del proyecto es, por tanto, más directa y sencilla.

Además del tablero de proyecto y los propios repositorios de Git, también se ha hecho uso de **Actions**, la herramienta de CI/CD de GitHub. Como su propio nombre indica, se definen acciones en flujos de trabajo que se ejecutan para los eventos del repositorio (p. ej., un nuevo *push*, la publicación de una nueva versión del producto, un *pull request*) que desee el desarrollador. Estas acciones pueden incluir, por ejemplo, la ejecución de pruebas, la construcción de artefactos o el despliegue del software sobre diversas plataformas. Así, se consigue automatizar los procesos de despliegue y se incorporan también medidas para el control de la calidad del código. En nuestro caso, en la ejecución de las pruebas se genera un informe sobre la cobertura



/ shipyard

Files					Coverage
crane	35	12	0	23	34.29%
node	236	204	0	32	86.44%
task	180	128	0	52	71.11%
db.py	4	4	0	0	100.00%
errors.py	8	8	0	0	100.00%
fields.py	12	9	0	3	75.00%
input_formats.py	10	4	0	6	40.00%
validators.py	12	12	0	0	100.00%
Folder Totals (8 files)	497	381	0	116	76.66%
Project Totals (15 files)	497	381	0	116	76.66%

Figura 1.3.: Análisis de la cobertura del código proporcionado por Codecov.

del código, el cuál se envía a **Codecov**, una plataforma especializada en este tipo de informes, para su análisis y visualización. En la figura 1.3 se muestra un ejemplo de la información que proporciona esta herramienta.

Los diagramas que se presentan a lo largo de este trabajo, incluidos los que componen el diseño de la herramienta desarrollada y sus componentes, se han creado con la herramienta online **draw.io**. Al integrarse directamente con Google Drive, se puede usar desde el navegador, siendo una solución muy cómoda y accesible para la creación de estos diagramas.

En cuanto al desarrollo y despliegue de la herramienta planteada, se han utilizado varias tecnologías. Los lenguajes de programación son Python y C, muy conocidos y utilizados en la industria. C es el lenguaje de referencia para la programación de sistemas y las tareas de bajo nivel, mientras que Python ofrece una sintaxis muy sencilla para la implementación de herramientas como aplicaciones o servidores. Python también posee una nutrida colección de librerías para distintos casos de uso, lo cual también fue muy atractivo para su elección. En este proyecto, se usan varias de estas librerías para facilitar la implementación de algunos aspectos de la herramienta. Destacamos las siguientes:

- **hug** - Escritura de APIs sencilla y sin ataduras. Esta librería trata de competir con otras más conocidas como Flask, proponiendo un modelo mucho más simple y dando más libertad al desarrollador.

- **click** - Más que una librería, se trata de un *framework* para la implementación de aplicaciones de línea de comandos (CLI). Ofrece todas las utilidades que se pueden

necesitar para este tipo de aplicaciones, como es la generación automática de páginas de ayuda.

- **marshmallow** - Esta librería permite la sencilla deserialización y serialización de objetos JSON a objetos Python. Esto es especialmente útil al trabajar con servicios web, ya que se pueden leer los datos JSON recibidos y validarlos frente a un esquema de datos definido por el desarrollador.

- **pymongo** - Ejecución de consultas sobre bases de datos MongoDB, que son las usadas en este proyecto.

- **paramiko** - Herramientas para trabajar con SSH desde Python. Esencial para el despliegue de las tareas sobre los nodos.

- **docker** - La librería oficial de Docker para Python. Con ella, se puede conectar con un servidor Docker (local o remoto) para ejecutar operaciones de construcción de imágenes o lanzamiento de contenedores.

- **requests** - Es la librería más conocida para la realización de peticiones HTTP en Python.

- **unittest** - Aunque no se trata de una librería externa como las demás, ya que forma parte de la librería estándar de Python, merece ser destacada por ser la utilizada para la escritura y ejecución de las pruebas del código desarrollado.

Además de estas librerías, se han usado otras para el análisis y formateo del código como son **pylint** y **autopep8**.

Como ya se ha mencionado, la herramienta desarrollada trabaja con **MongoDB** para la gestión del almacenamiento persistente de los datos y el acceso a los mismos. Se trata de un gestor de bases de datos documental que pertenece a la familia NoSQL (*Not only SQL*). Las bases de datos documentales divergen del modelo relacional tradicional que siguen otros gestores tan famosos como MySQL o PostgreSQL. No existen los conceptos de tabla o fila, sino que se almacenan colecciones de documentos no estructurados. En el caso de MongoDB, se trabaja con documentos JSON, formato ampliamente usado en el entorno web para la transmisión de datos debido a su legibilidad y velocidad de serialización/deserialización. Cuando decimos que los documentos son no estructurados, nos referimos a que no tienen por qué poseer los mismos campos, ya que no hay un esquema definido a seguir. MongoDB no cumple con ACID, pero es una pérdida aceptable a cambio de la flexibilidad y facilidad de uso que nos ofrece. Además, el lenguaje de consultas es mucho más claro e intuitivo que SQL, permitiendo realizar consultas de envergadura similar.

La idea principal en torno a la cual gira este trabajo es la del uso de contenedores para el despliegue de tareas de tiempo real. En este sentido, **Docker** es la tecnología de contenerización en la que nos hemos apoyado para la implementación de la herramienta propuesta. Esta elección recae principalmente en el hecho de que se trata del motor de contenedores más conocido y utilizado. De la mano de Docker, también se ha utlizado **Docker Compose** para el despliegue sencillo de un entorno de desarrollo completo que permita replicar un despliegue de producción de forma simplificada. Las imágenes de contenedores definidas en este proyecto, se han alojado en el repositorio de imágenes **DockerHub**, aprovechando que proporciona un sistema de

Figura 1.4.: Raspberry Pi 4B utilizada en el trabajo.

construcción automática de imágenes.

Para realizar las pruebas del sistema desarrollado, se ha usado una **Raspberry Pi 4B**, concretamente la que se muestra en la figura 1.4. Se trata de un SBC (*Single Board Computer*) de bajo coste y muy versátil, pudiendo usarse para infinidad de aplicaciones. Las características detalladas de esta plataforma se pueden ver en la tabla 1.1.

Procesador	Cortex-A72
Arquitectura	ARMv7
Número de núcleos	4
Frecuencia de reloj	1500 MHz
RAM	4 GB
Sistema operativo	Raspberry Pi OS Lite (Linux 4.19)

Tabla 1.1.: Especificaciones de la Raspberry Pi usada en el trabajo.

El sistema operativo de este dispositivo es GNU/Linux (concretamente, basado en Debian), al que se le ha aplicado la revisión 24 del parche de tiempo real `PREEMPT_RT`, obteniendo finalmente un kernel con versión `4.19.71-rt24-v7l+`. Este parche aporta al sistema la capacidad de realizar planificación de procesos apropiativa. En el apéndice B se explica con detenimiento el proceso seguido para usar el kernel de Linux con `PREEMPT_RT` en la Raspberry Pi.

1.5. Estructura de la memoria

En el capítulo 1, se ha introducido el proyecto y explicado tanto la motivación detrás del mismo como los objetivos planteados, la planificación seguida y las herramientas, tecnologías y metodologías que se han utilizado.

En el capítulo 2, se realiza un estudio en profundidad del estado de la técnica en lo relativo a los sistemas confiables, la computación en la nube y sus derivados, las tareas de tiempo real y las tecnologías de virtualización. Se introducen todos los conceptos relevantes, además de presentar otros trabajos interesantes llevados a cabo en estas áreas.

En el capítulo 3, se analiza el rendimiento de los procesos de tiempo real contenerizados, haciendo especial hincapié en el impacto que tienen sobre el mismo las tecnologías de contenerización frente a procesos no virtualizados.

En el capítulo 4, se aborda el diseño de una herramienta para el despliegue y seguimiento de tareas de tiempo real en entornos distribuidos, así como su implementación y prueba. Se presentan los resultados obtenidos y se discute sobre los mismos.

En el capítulo 5, se realiza una revisión del proyecto completo y se presentan las conclusiones, así como las maneras en las que se puede extender y mejorar el trabajo realizado.

2. Revisión del estado de la técnica

Antes de poder aventurarnos el desarrollo de una herramienta para la orquestación de tareas de tiempo real usando contenedores, es necesario estudiar y comprender los fundamentos y peculiaridades de diversos campos como son los sistemas de tiempo real, la planificación de procesos, la computación en la nube o las tecnologías de virtualización. En este capítulo, se introducen los conceptos más importantes de estos campos, además de revisar otros trabajos realizados con el objetivo de comprobar cuál es el estado de la investigación en dichos temas.

2.1. Sistemas empotrados, confiables y de criticalidad mixta

Los ordenadores y teléfonos inteligentes que usamos a diario nos permiten realizar multitud de tareas diferentes: desde leer el correo o navegar por internet hasta editar imágenes y vídeos o realizar videollamadas. Aunque no nos demos cuenta, interactuamos habitualmente con muchos más sistemas informáticos además de los ya mencionados. La computación está presente en los coches, los trenes, los aviones, los satélites espaciales, los televisores o las lavadoras. Estos sistemas, que reciben el nombre de sistemas empotrados, son diseñados para llevar a cabo de manera óptima un conjunto de tareas específico, frente al enfoque generalista de los ordenadores de uso personal. Como se introduce en [2], los sistemas empotrados son comunes en contextos en los que el rendimiento es primordial, como son las comunicaciones de red o la compresión/decompresión de audio y vídeo para retransmisiones en directo. En este libro se exponen diversas aplicaciones de los sistemas empotrados de alto rendimiento para sistemas ciber-físicos o CPS (*Cyber-Physical Systems*). Los CPS son, en esencia, sistemas informáticos que interactúan con procesos físicos, actuando en función de los cambios en su entorno. Este aspecto hace que el diseño y la implementación de los CPS difiera considerablemente del resto de sistemas empotrados, ya que hay que prestar especial atención a las características del entorno en el que se desplegará el sistema y a los mecanismos de entrada y salida (interacción con el entorno), además de optimizar el consumo de memoria y procesamiento para cumplir con las limitaciones del hardware [3].

En algunos casos, estos sistemas son críticos, lo que significa que un fallo en su funcionamiento supone daños graves a las personas o al medio ambiente. Este es el caso de los aviones, los trenes o las plantas nucleares, por ejemplo. En estos sistemas, cobra especial importancia el concepto de confiabilidad, es decir, garantizar el correcto funcionamiento del sistema en todo momento. En la figura 2.1 se pueden apreciar los atributos que debe poseer un sistema confiable. A nivel de software, existen arquitecturas y patrones de diseño orientados a garantizar la tolerancia ante los fallos del mismo [4][5]. Por otra parte, la replicación de componentes [6] es una técnica muy usada tanto para el software como el hardware. Lo que se intenta con la replicación es asegurar que un cálculo o procesamiento se lleva a cabo de forma correcta, aunque alguna de las réplicas falle. En [7], se realiza una revisión de los sistemas de control tolerantes ante fallos dividiéndolos en tres tipos: AFTCS (*Active Fault Tolerant Control Systems*), PFTCS (*Passive*

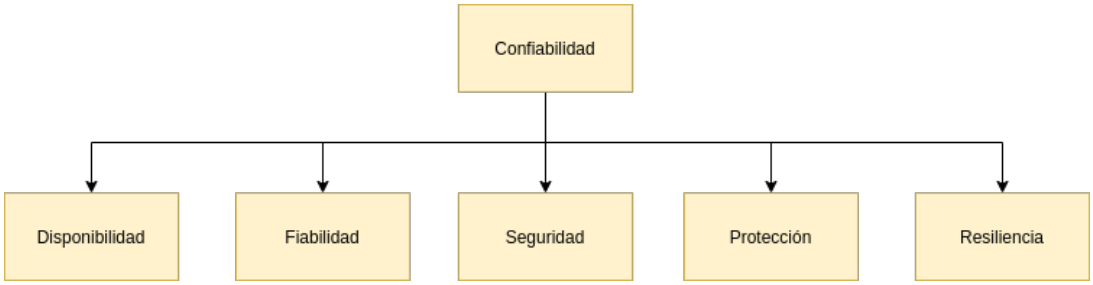


Figura 2.1.: Principales aspectos de la confiabilidad.

Fault Tolerant Control Systems) o HFTCS (*Hybrid Fault Tolerant Control Systems*). Para cada uno de estos tipos, los autores presentan las principales arquitecturas usadas, los modelos de análisis matemático usados para su validación y las últimas técnicas usadas para su diseño. Por otra parte, el estudio realizado en [8] tiene como objetivo analizar la aplicación de los estándares de seguridad, protección y privacidad al diseño y desarrollo de sistemas confiables, llegando los autores a la conclusión de que cada vez están ganando más popularidad los de protección y privacidad, aunque los procesos para asegurar estos aspectos son menos maduros que los relacionados con la seguridad. Además, también se identifica una falta de acción combinada en estos aspectos, trabajando normalmente por separado en cada uno de ellos.

Para muchos sistemas de control, el tiempo es también un aspecto relevante a la hora de garantizar su correcto funcionamiento. Estos son los llamados sistemas de tiempo real. En un sistema normal, en el que se obtiene una salida al aplicar una cierta lógica sobre la entrada, la corrección de dicha salida depende de la corrección de la lógica aplicada, es decir, del cálculo realizado. No obstante, cuando hablamos de sistemas de tiempo real, la corrección de la lógica no es suficiente para determinar que el funcionamiento es correcto, sino que esta corrección depende también del momento temporal en el que se obtiene [9]. En otras palabras, si el cálculo es correcto pero el resultado no llega en el momento en el que se necesita, se considera como un fallo del sistema. Las tareas que ejecuta un sistema de tiempo real están, por tanto, sujetas a restricciones temporales. Normalmente, hablamos de un tiempo límite o *deadline* antes del cuál se debe haber completado la tarea. Comúnmente, se suelen clasificar las tareas en dos tipos dependiendo de las consecuencias de incumplir con sus restricciones temporales:

- Tareas de tiempo real «blandas»: Se denominan así aquellas tareas en las que no completar las mismas antes del límite supone una reducción en la calidad del servicio (QoS).
- Tareas de tiempo real «duras»: En estas tareas, el no cumplir con las restricciones temporales supone un fallo grave del sistema con posibles consecuencias catastróficas.

Para implementar estos sistemas, se hace uso de herramientas y algoritmos de planificación de procesos específicos que permiten la ejecución de estos tipos de tareas cumpliendo con sus restricciones temporales. Esto se explica en más detalle en la sección 2.4.

En este trabajo, nos centraremos especialmente en los sistemas de control aplicados a entornos industriales. Debemos diferenciar entre dos conceptos: PCS (*Process Control System*), que es el sistema encargado de controlar una parte concreta de la planta (p. ej., una turbina o un brazo robótico), e ICS (*Industrial Control System*), que se refiere al control de toda la

planta al completo. Un ICS está compuesto, por tanto, de múltiples PCS que se encargan de controlar los distintos aspectos del proceso de producción. Estos PCSs son sistemas empotrados como los que hemos comentado a los que se aplican también los conceptos de confiabilidad y seguridad. En [10] se realiza un estudio del estado del arte en cuanto a la seguridad de los ICS, concluyendo que, aunque las vulnerabilidades de muchos de los sistemas actuales son conocidas, la aplicación de parches que las solucionen son inviables debido a los altos costes asociados a la recertificación o directamente debido a la incompatibilidad con algunos sistemas más antiguos. Es necesario, por tanto, incorporar los conceptos de seguridad a los procesos de diseño desde el primer momento para evitar dar lugar a estas situaciones.

En 2007, Vestal [11] publica una primera propuesta para la planificación de conjuntos de tareas de criticalidad mixta. Este importante avance es considerado por muchos como el inicio de la investigación en sistemas de criticalidad mixta o MCS (*Mixed-Criticality Systems*). La idea detrás de estos sistemas es, como su propio nombre indica, poder ejecutar sobre una misma plataforma hardware tanto tareas críticas como tareas no críticas o con niveles de criticalidad menores, asegurando en todo momento que las restricciones temporales se cumplen, al menos para las tareas más críticas. Desde entonces, se han planteado muchos modelos para la implementación de estos sistemas. En [12], Burns realiza una revisión de toda la investigación realizada en este campo hasta marzo de 2019. Se muestran en esta revisión algunas arquitecturas propuestas, además de las principales técnicas de análisis para estos sistemas en plataformas uniprocesador y multiprocesador. Burns identifica la conciliación entre la separación de los procesos y la compartición de los recursos como el principal problema de los MCS. En este aspecto, nuestro trabajo propone el uso de contenedores como medio de ejecución de los distintos procesos sobre una misma plataforma consiguiendo esa separación.

No podemos terminar nuestra revisión de los sistemas empotrados sin hablar sobre el uso del kernel de Linux para su implementación. Como referencia, hemos tomado dos estudios realizados en 2004 sobre el uso de Linux para sistemas empotrados [13][14]. En el primero, se destaca la buena situación de Linux en este campo, suponiendo las soluciones comerciales basadas en el kernel de Linus Torvalds el 15,5 % del mercado. El segundo estudio es una encuesta realizada a 268 personas que trabajan en el campo de los sistemas empotrados, ya sea académicamente o de forma comercial. Lo más destacable es que la mayoría de los participantes usan Linux en sistemas empotrados para comunicaciones, dispositivos móviles o control de maquinaria. Esto último es especialmente alentador para este trabajo, en el que pretendemos usar Linux como base para la implementación de ICS.

2.2. La nube en los sistemas industriales: fog/edge computing para sistemas de tiempo real

Desde su aparición alrededor de 2006, la computación en la nube o *cloud computing* ha experimentado un crecimiento abismal, cambiando completamente la manera en la que las organizaciones gestionan su infraestructura y en la que los usuarios acceden a servicios. Atendiendo a la definición del NIST [15], se trata de un modelo de prestación de servicios que ofrece acceso ubicuo, prácticamente ilimitado y bajo demanda a un conjunto de recursos de computación (p. ej., almacenamiento, tiempo de procesamiento, comunicaciones), todo ello a través de internet. Sus principales características son:

- Acceso bajo demanda: El consumidor es el que decide qué recursos necesita y en qué cantidad, sin necesidad de intervención humana por parte del proveedor del servicio.

- Consumo a través de internet: El acceso a estos recursos se realiza mediante los protocolos ya conocidos y bien establecidos que potencian la web (p. ej., HTTP), permitiendo así un acceso ubicuo.

- Acceso compartido: Los recursos del proveedor se uniformizan, formando una especie de bolsa de recursos que es usada por múltiples consumidores (*multi-tenant*). Por ejemplo, aunque desde el punto de vista de un usuario pueda parecer que tiene una CPU para él solo, en realidad la CPU física es usada por varios usuarios a la vez.

- Elasticidad: Dependiendo de la demanda, se pueden asignar o liberar los recursos de forma dinámica, consiguiente sistemas más eficientes y capaces de dar respuesta a cargas de trabajo más altas.

- Servicio medido: El uso de los recursos se monitoriza de forma automática, lo que proporciona transparencia tanto para el consumidor como el proveedor.

A raíz de la propuesta de la computación en la nube, han surgido varios modelos de prestación de servicios a usuarios. Según el tipo de recurso que ofrecen, estos modelos se pueden clasificar principalmente en SaaS (*Software as a Service*), PaaS (*Platform as a Service*) o IaaS (*Infrastructure as a Service*). Desde el punto de vista de los usuarios, estos modelos tienen el beneficio de que permiten pagar solo por lo que necesitas en cada momento, adaptándose perfectamente a tus necesidades. Además, estos usuarios ya no se ven limitados por el hardware que poseen. Prácticamente cualquier ordenador es capaz de conectarse a internet y, de esta forma, acceder a unos recursos prácticamente ilimitados en la nube. Esto es especialmente evidente para las empresas, las cuáles ya no necesitan mantener su propia infraestructura de hardware para soportar su operación diaria, con los enormes costes y complicaciones que esto supone.

En el campo del control industrial, en el que se centra este trabajo, podríamos pensar en la nube como una plataforma ideal para procesar las enormes cantidades de datos que producen las plantas e implementar las técnicas de aprendizaje automático necesarias para mejorar su autonomía. No obstante, existe un gran problema: la latencia. En el modelo de la nube, todo el procesamiento está centralizado en un centro de datos[1], el cuál se encuentra lejos de la planta donde se recogen los datos. Enviar estos datos a la nube y esperar una respuesta supone demasiado tiempo como para poder considerar delegar en ella tareas de control de la operación de la planta. En entornos controlados, donde el centro de datos se encuentra relativamente cerca de la planta, las pruebas llevadas a cabo en [16] indican que la latencia puede ser lo suficientemente predecible para la implementación de ciertos sistemas de control con requisitos temporales suaves. Sin embargo, se trata de una situación poco factible, ya que es inviable ubicar un centro de datos cerca de todas las instalaciones industriales. En [17], se analiza si los ICS están preparados para ser desplazados a la nube, pero centrándose más en el aspecto de la seguridad de estos sistemas. El autor identifica que, aunque la nube es una plataforma muy robusta, también son flagrantes las dudas que produce en cuanto a su seguridad frente a ataques. Por estas razones, no consideramos que la nube sea un paradigma adecuado para dar respuesta a los nuevos requisitos de la industria.

[1] Normalmente, existen varios centros de datos distribuidos geográficamente, pero a efectos del problema que se plantea, se sigue viendo como una «centralización».

Aunque la nube no sea viable, bien es cierto que las ideas de elasticidad y ubicuidad que plantea son interesantes para el problema en cuestión. Al final del día, si queremos realizar análisis en tiempo real de los datos que generan los procesos industriales para tomar decisiones, necesitamos más potencia computacional de la que ningún dispositivo individual nos puede ofrecer. ¿La solución? Aprovechar los recursos de computación de todos los dispositivos ya presentes en las plantas industriales. De acercar estas ideas de la nube a los dispositivos del borde de la red surgen dos nuevos paradigmas de computación: *fog* y *edge computing*. En estos paradigmas, se pretende acercar el procesamiento que se realiza sobre los datos a los propios dispositivos que generan dichos datos. En el contexto industrial, esto también supone estar más cerca de los dispositivos que deben actuar en respuesta a este procesamiento, con lo que se obtendrían tiempos de respuesta mejores que con la nube. La diferencia entre *fog* y *edge* radica en lo cerca del borde de la red que ubiquemos el procesamiento de los datos, si bien es cierto que en algunos escritos se usan los términos *fog* y *edge* de manera intercambiable [18], sin diferencia aparente entre ambos.

- En *fog*, se trata de nodos ubicados en la misma red local que las fuentes de datos.
- En *edge*, los propios dispositivos que generan los datos ya realizan un cierto procesamiento de los mismos.

En la figura 2.2 se puede apreciar como estos dos paradigmas, junto con la nube, constituyen una arquitectura por capas, donde cada uno de ellos tiene unas características concretas. Se trata, por tanto, de modelos complementarios, no exclusivos. En la capa *edge*, donde se generan los datos, se pueden implementar los mecanismos de control con restricciones más duras, ya que se tienen unos tiempos de respuesta más rápidos al actuar directamente en base a los datos generados. En la capa *fog*, los nodos pueden realizar tareas de análisis de datos más exigentes, dado que poseen más capacidad de computación que los dispositivos del *edge*. También se pueden implementar aquí tareas de control. Por último, la nube recibiría todos los datos producidos por la planta, además de otras plantas que también posea la organización, realizando la agregación y el análisis en profundidad de los datos, obteniendo informes que puedan ayudar a los supervisores en la identificación de fallos y la mejora del rendimiento.

Dejando atrás la ya explicada nube, vamos a adentrarnos más en el *fog*. Esta capa se compone de los llamados nodos *fog*, que son los encargados de llevar a cabo las distintas tareas. Cualquier dispositivo con capacidad de procesamiento, memoria y conexión de red puede ser un nodo viable. De esta forma, se pueden usar como nodos routers, switches y otros dispositivos de redes, así como cámaras de videovigilancia. Se aprovecha la capacidad de procesamiento de todos estos dispositivos cuando no se están usando al 100 %. Como ya se ha indicado, estos nodos se encuentran ubicados en la misma red local (LAN) que los dispositivos del borde de la red que generan los datos. Las distintas tareas de análisis y control a realizar sobre estos datos se distribuyen sobre los nodos *fog* de forma dinámica dependiendo de la carga de trabajo que posean los mismos. A la hora de asignar una tarea a un nodo, siempre se debe intentar hacer de forma que el nodo elegido esté lo más cerca posible de la fuente de los datos [19]. Esto es especialmente importante para tareas de control, donde es posible que no todos los nodos *fog* de la planta tengan la latencia necesaria para cumplir con los requisitos temporales. En este sentido, en [20] se propone el uso de TSN (*Time-Sensitive Networking*) para las comunicaciones en la capa *fog*, con el objetivo de reducir estas latencias. Este modelo de computación distribuida sobre una bolsa o *pool* de recursos es muy similar al que encontramos en la nube, con la diferencia de que en el *fog* estos recursos son mucho más limitados, por lo

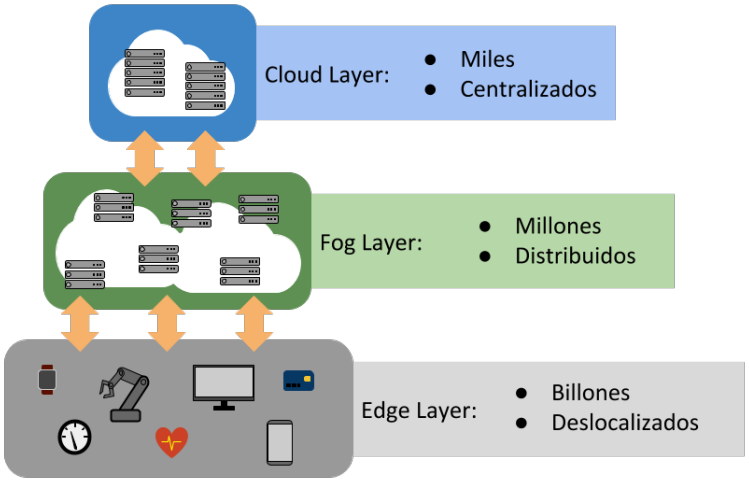


Figura 2.2.: Estructura por capas *cloud-fog-edge*.

que es también esencial optimizar el uso de los mismos. Debido a esta similitud con el modelo de la nube, las mismas tecnologías de virtualización usadas en ésta también son útiles en el nuevo paradigma [21]. Algunos modelos ya han sido propuestos para la aplicación del *fog* a los sistemas industriales y el modelado de la Industria 4.0 [22] [23].

Aunque la capacidad de procesamiento del *fog* es mucho menor que la de la nube, sigue siendo útil para realizar algunas tareas menos intensivas evitando tener que desplazar todos los datos a la nube, con la latencia y la carga en la red que eso supone. Como consecuencia del avance del IoT (*Internet of Things*) industrial, que es considerada una de las tecnologías más importantes para la Industria 4.0 [24], el número de dispositivos presentes en las plantas y, por tanto, la cantidad de datos que se generan va a ser mucho mayor, poniendo aún más presión sobre la red, que es donde se podría producir el cuello de botella [25]. El *edge* trata de solucionar esto al reducir la cantidad de datos que se deben enviar hacia las capas superiores procesando una parte de ellos de forma local. Cabe destacar que el *fog* también permite relajar el tráfico de red [26]. Los dispositivos del borde de la red, aún ofreciendo menos potencia que la combinada del *fog*, también pueden ser usados para realizar algunas acciones sobre los datos. En concreto, las tareas de control que requieran de un tiempo de respuesta más rápido pueden realizarse en esta capa, tomando decisiones directamente sobre los datos que recoge el dispositivo. Por otra parte, Khan [27] identifica la falta de confianza en los sistemas *edge* como uno de los principales problemas a los que se enfrenta el paradigma, además de la integración de sistemas y la movilidad. Esta falta de confianza es especialmente grave si pretendemos desplegar sistemas de control críticos sobre plataformas de este tipo. Como solución al problema de la confianza, Stanciu [28] propone usar tecnología de *blockchain* para desplegar sistemas de control distribuidos en el *edge*.

Sectores como el automovilístico, energético o de alimentación están muy interesados en el desarrollo de la Industria 4.0 debido a los importantes beneficios que se estima que puede tener para sus procesos. Gracias al 5G y a las mejoras en las comunicaciones, el IoT se convierte en una idea capaz de dar a las organizaciones una cantidad de información sobre sus procesos industriales jamás vista antes. Estos datos son la base de la Industria 4.0 [29], ya que permiten

la definición de sistemas ciber-físicos y gemelos digitales que replican perfectamente los procesos físicos en el plano digital, lo que abre la puerta a un control inédito sobre dichos procesos. Además de las comunicaciones, otro reto muy importante para la consecución de estos objetivos es el modelado de estos sistemas cuidando los aspectos deseados de seguridad y fiabilidad [30]. En este sentido, creemos que el *fog* es un modelo de computación que puede facilitar el diseño de estos CPS al proporcionar una plataforma robusta y unificada sobre la que implementarlos, siempre y cuando se pueda garantizar el determinismo en la ejecución de las tareas críticas.

2.3. Tecnologías de virtualización: hipervisores y contenedores

De manera simple, la virtualización consiste en abstraer ciertas capas de un sistema convencional (p. ej., el hardware). Sobre estas abstracciones se pueden ejecutar procesos o sistemas completos de manera aislada. Este aspecto es el que hace de la virtualización un concepto tan relevante para la computación en la nube, ya que posibilita el acceso compartido a los recursos. Por ejemplo, sobre una misma CPU se podrían ejecutar varios sistemas operativos completamente funcionales de forma que ninguno de ellos sepa de la existencia de los otros. Cada uno de estos sistemas cree que tiene una CPU completa para él gracias a la abstracción del hardware. Aunque las tecnologías de virtualización han experimentado un empuje enorme en los últimos años debido a la expansión de la nube, no se trata de una idea novedosa. En 1974, Popek y Goldberg [31] definían los requisitos que debía cumplir una plataforma para poder virtualizar sistemas completos. Hoy en día, hablamos de virtualización en dos niveles diferentes: a nivel del hardware (máquinas virtuales) o a nivel del sistema operativo (contenedores).

Lo que se quiere decir con virtualización a nivel del hardware, es que es éste el que se abstrae. Esto es precisamente lo que sucede con las máquinas virtuales. El hardware «real» se puede fraccionar para asignarlo a las diversas máquinas virtuales, que lo usan como si se tratase de una plataforma física normal y corriente. Así, se puede regular la cantidad de CPU, memoria RAM o, incluso, acceso a dispositivos de I/O que tiene cada máquina virtual. Todo esto es posible gracias a los hipervisores, que son la pieza de software encargada de abstraer el hardware y regular su uso por parte de las máquinas virtuales. Dependiendo de la relación existente entre el hipervisor y el hardware que abstraen, nos encontramos con hipervisores de tipo 1 y 2. Un hipervisor de tipo 2 se ejecuta sobre un sistema operativo tradicional, que recibe el nombre de anfitrión, mientras que los de tipo 1 prescinden del anfitrión, ejecutándose directamente sobre el hardware que gestionan (*bare metal*). En la figura 2.3 se representa gráficamente esta diferencia.

Como es obvio, los hipervisores de tipo 1 suponen una solución mucho más liviana que los de tipo 2 al no tener un anfitrión. Por otra parte, pueden asignar prácticamente todos los recursos hardware de la plataforma a las máquinas virtuales de forma dinámica, mientras que los de tipo 2 se ven restringidos a los límites impuestos por el anfitrión. Los hipervisores de tipo 2 son más utilizados en el ámbito personal, con herramientas como VirtualBox o VMWare siendo muy utilizadas para la prueba de software, mientras que los de tipo 1 son los que predominan a nivel empresarial e industrial, con claros ejemplos como PikeOS o Xen. Este último supuso un importante avance en la ejecución segura y aisla de máquinas virtuales sobre hardware de uso común [32]. En lo que a sistemas de tiempo real se refiere, RT-Xen [33] se presentó como un

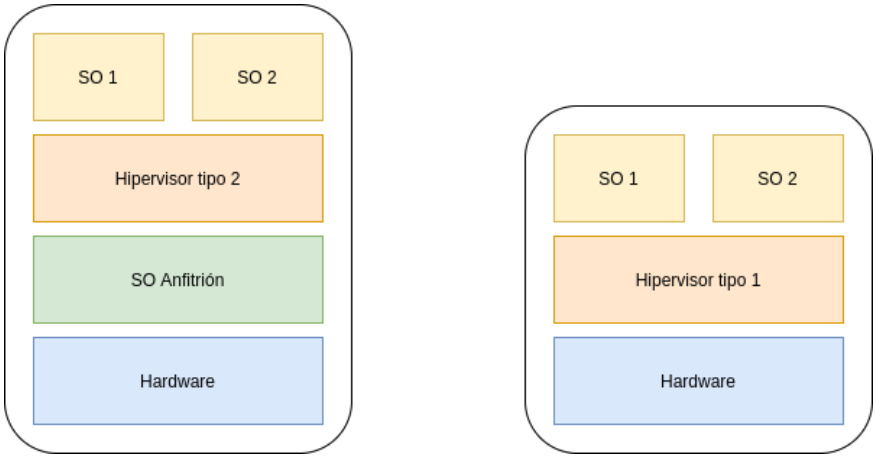


Figura 2.3.: Capas de la virtualización mediante hipervisores de tipo 1 y 2.

marco de referencia para la planificación de procesos con restricciones temporales en sistemas Linux virtualizados usando Xen, obteniendo prometedores resultados.

Mientras que en la virtualización a nivel del hardware las máquinas virtuales contienen un sistema operativo completo, en la virtualización a nivel del software solo se virtualizan procesos. Estos procesos virtualizados, que reciben comúnmente el nombre de contenedores, acceden a los servicios básicos del sistema operativo anfitrión. De forma análoga a los hipervisores, los motores de contenedores son los encargados de gestionar los procesos y su acceso al sistema operativo que comparten. Realmente, los procesos contenerizados se ejecutan sobre el sistema anfitrión como lo harían los procesos «nativos». La única diferencia se encuentra en que los procesos contenerizados están aislados del resto. Estos procesos solo pueden ver al resto de procesos de su mismo contenedor, no los del resto de contenedores ni los no contenerizados. Para conseguir este aislamiento, los motores de contenedores aprovechan varias características del kernel de Linux, entre las que destacamos:

- Espacios de nombres (*namespaces*): los identificadores de proceso (PID), las interfaces de red o la parte del sistema de ficheros que puede ver un proceso contenerizado depende del espacio de nombres al que se asigne cuando se crea. Usando esta herramienta, se puede restringir lo que ven los contenedores del sistema anfitrión.

- Grupos de control (*control groups*): Los `cgroups` son mecanismos que permiten regular el acceso al hardware por parte de los contenedores, así como su acceso compartido.

A primera vista, cabría pensar que, dado que contienen un sistema operativo completo, las máquinas virtuales son mucho más pesadas que los contenedores. No obstante, en trabajos como [34] [35] se desmiente esta percepción. Concretamente, en el último se compara el popular motor de contenedores Docker con KVM, un famoso hipervisor para Linux, llegando a la conclusión de que ninguno de los dos penaliza de manera significativa el rendimiento en cuanto a uso de CPU o memoria, si bien es cierto que las máquinas virtuales de KVM obtenían peores resultados en las operaciones de I/O. Por otro lado, la seguridad en los contenedores es un aspecto todavía en desarrollo [36], sobre todo en la seguridad del kernel compartido.

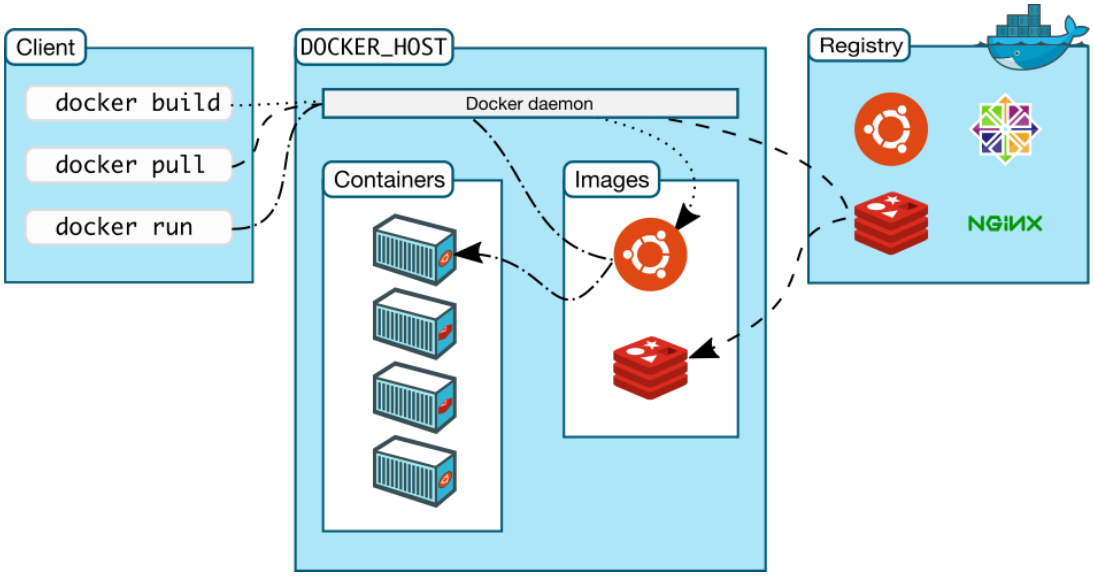


Figura 2.4.: Esquema de funcionamiento de Docker.

Existen múltiples motores de contenedores usados ampliamente en la actualidad, como es el caso de Docker, Rocket (rkt) o LXC (*LinuX Containers*). Kozhirbayev y Sinnot [37] realizaron en 2017 una comparativa del rendimiento entre Docker y LXC[2]. Los resultados obtenidos coinciden con los vistos en [35] en tanto que los procesos contenerizados no tienen una penalización en el consumo de CPU o RAM frente a los nativos, aunque sí que se observa esta penalización en las operaciones de I/O. Como el rendimiento es similar para todos los motores de contendores, se ha decidido poner el foco sobre Docker para este trabajo debido a que es el más popular, con el consecuente impacto que esto tiene en la calidad del soporte y de la documentación disponible.

En Docker, al igual que en muchos otros motores de contenedores, los contenedores se lanzan a partir de imágenes, definidas mediante un `Dockerfile`. Una imagen es reutilizable y universal, pudiendo lanzar réplicas de un mismo contenedor en cualquier plataforma soportada por la imagen. Para trabajar con imágenes y contenedores, el sistema Docker se compone de dos partes principales: el demonio[3] y el cliente. El demonio es el encargado de realizar todas las soperacones relacionadas con imágenes o contenedores, incluyendo su construcción, lanzamiento y gestión posterior. Para realizar estas acciones, expone una API con la que pueden interactuar los distintos clientes, siendo el oficial que proporciona Docker una aplicación de línea de comandos. A estos dos componentes se podría añadir un tercero, el registro de imágenes. Como su propio nombre indica, se trata de una especie de base de datos de imágenes listas para su uso. De esta forma, se pueden almacenar y compartir imágenes entre varios dispositivos a través de la red. No obstante, se puede hacer uso de Docker de manera local sin necesidad de usar un registro externo. Un esquema del funcionamiento de Docker que se acaba de explicar se puede ver en la figura 2.4.

Centrando nuestra atención de nuevo en los sistemas de control industrial, se plantea la po-

[2]Concretamente, se hace uso de Flockport.
[3]Un demonio o *daemon* es un tipo de proceso que se ejecuta en segundo plano para ofrecer un servicio.

sibilidad de implementarlos usando contenedores [16], aunque los principales inconvenientes identificados son los relacionados con la seguridad. En [38], los autores proponen un modelo de planificación de procesos contenerizados para conjuntos de tareas con distintos niveles de criticalidad, defendiendo el uso de contenedores por su capacidad de aislamiento. Sin embargo, en la encuesta sobre contenerización para tiempo real realizada por Struhár et al. en 2020 [39], se llega a la conclusión de que son necesarias mejores herramientas para la gestión de contenedores de tiempo real. Por ello, en este trabajo se plantea el desarrollo de una herramienta para este fin como prueba de concepto. Los autores del estudio destacan también la falta de un mecanismo determinista de comunicación entre procesos contenerizados y de herramientas de validación y prueba, además de los problemas de seguridad ya mencionados.

2.4. Aspectos de computación en tiempo real: RTOS, algoritmos de planificación y herramientas

En esta sección, se va a explicar con mayor detalle el funcionamiento de los sistemas de tiempo real ya mencionados en las secciones anteriores. En estos sistemas, el tiempo es un factor crucial a la hora de determinar la validez de las salidas. De esta forma, las tareas ejecutadas se ven limitadas por restricciones temporales. Dependiendo de la severidad de las consecuencias del incumplimiento de las restricciones, solemos hablar de tareas y restricciones duras o blandas. Las restricciones duras son aquellas que, si se incumplen, tienen consecuencias son muy graves, como pueden ser daños a la vida de las personas en el caso de sistemas que sean críticos. En cuanto a las restricciones blandas, solo se experimenta una pérdida de calidad en el servicio cuando se incumplen, aunque se suele limitar el número de incumplimientos permitido. En la figura 2.5 se muestra la diferencia entre los dos tipos de restricciones según su función de utilidad [40], que es la que define el valor de la salida del sistema en función del tiempo. Como se puede apreciar, el resultado de una tarea dura solo es útil si se da entre el inicio de la tarea y su límite temporal fijado, provocando daños de cualquier otra manera. En las tareas blandas, el resultador puede tener valor incluso si se obtiene después del límite, aunque se va reduciendo con el tiempo.

Debido a la gran responsabilidad que recae sobre el cumplimiento de estos requisitos, los sistemas empotrados sobre los que se ejecutan estas tareas tienen características diferentes a las de los sistemas de uso general. A nivel del hardware, por ejemplo, las CPU tienen un diseño más simple que haga más fácil verificar el determinismo en sus operaciones. En [41], Rotenberg y Anantaraman describen el funcionamiento de las CPU de alto rendimiento y las comparan con las empotradas. La ejecución predictiva, por ejemplo, es severamente limitada en los sistemas empotrados. Por otro lado, las operaciones de acceso a caché y RAM suponen están optimizadas para necesitar un menor número de ciclos. Normalmente, el conjunto de instrucciones (ISA) de estas CPU es reducido, conocido como RISC, reduciendo la complejidad y facilitando la programación a bajo nivel. ARM es una de las arquitecturas de microprocesador basadas en RISC más usadas para el diseño de sistemas empotrados. Otro aspecto diferencial es el número de núcleos o *cores* presentes en los procesadores. Poseer varios núcleos hace posible la ejecución en paralelo de instrucciones, mejorando el rendimiento en los sistemas. No obstante, este paralelismo dificulta asegurar el determinismo, aunque algunos algoritmos de planificación de procesos han sido propuestos [42] y la investigación es activa en este área. Para asegurar además que las tareas se puedan ejecutar correctamente, los sistemas de tiempo

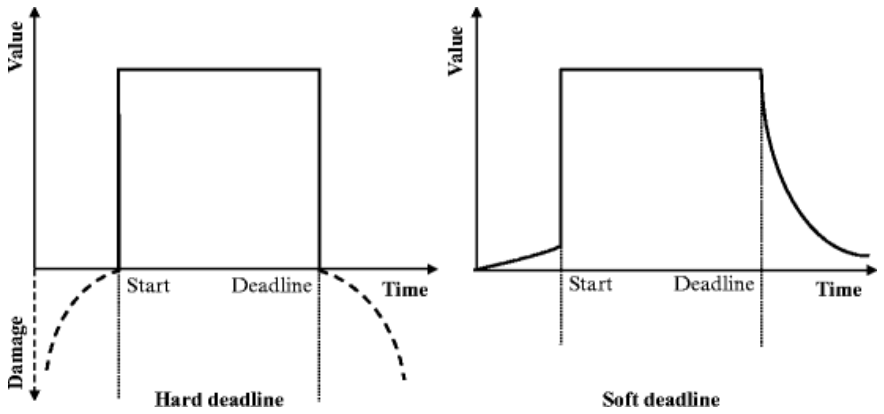


Figura 2.5.: Funciones de utilidad para restricciones temporales duras y blandas.

real suelen dejar capacidad de CPU extra libre.

Además del hardware, el software usado para implementar estos sistemas también es especial. Los lenguajes y herramientas deben proporcionar al programador ciertas comodidades para la implementación de procesos en tiempo real [43]:

- Especificar momentos en los que las acciones se deben realizar.
- Especificar los momentos antes de los cuales las acciones deben terminar.
- Permitir la ejecución repetitiva (periódica y aperiódica) de trabajo.
- Limitar la varianza en las operaciones de entrada y salida.
- Responder a situaciones donde no todas las restricciones temporales se pueden cumplir.
- Responder a situaciones en las que los requisitos de tiempo cambian de manera dinámica.

Una parte muy importante de los sistemas empotrados es su sistema operativo, el cual gestiona los recursos físicos de la plataforma y controla el uso de los mismos por parte de los procesos implementados por los desarrolladores. Al igual que ocurre con el hardware, estos sistemas operativos también tienen características diferentes a los de propósito general para poder dar cobijo a tareas con restricciones temporales, recibiendo el nombre de RTOS (*Real Time Operating System*). La principal diferencia, podríamos decir que se encuentra en los algoritmos de planificación de tareas usados, aunque cabe destacar que también hay otras áreas importantes como el acceso a memoria o la gestión de las interrupciones. El algoritmo de planificación es, a grandes rasgos, la política que decide en qué orden se ejecutan las tareas del sistema. En los sistemas normales, normalmente se asigna a cada proceso una fracción igualitaria de tiempo de CPU, ejecutándose hasta que consume su fracción de tiempo o se bloquea (p. ej., por una operación de lectura o escritura). En los sistemas de tiempo real, este método no es factible, ya que los procesos tienen distintas prioridades y no pueden quedar a la espera de que otro proceso de menor prioridad libere la CPU. Por ello, la planificación de procesos de tiempo real es apropiativa (*preemptive scheduling*), de forma que el sistema operativo puede quitar la CPU

a un proceso que está en ejecución para asignársela a otro más prioritario. Como se destaca en [44], la planificación de procesos óptima para un sistema de tiempo real «duro» depende enormemente del problema en cuestión. ¿Se trata de un sistema uniprocesador o multiprocesador? ¿Todas las situaciones son conocidas de antemano? ¿Las tareas son periódicas o aperiódicas? Dependiendo de la situación, podemos encontrarnos con que un algoritmo de planificación es más eficaz que otro.

Antes de entrar a explicar algunos de los algoritmos de planificación de tareas más importantes, debemos entender mejor como son estas tareas. Las tareas en un sistema de tiempo real pueden ser provocadas por tiempo o por eventos. Las tareas provocadas por tiempo son tareas periódicas, con un tiempo definido para sus ciclos. Las caracterizamos con los siguientes atributos:

- Tiempo de ejecución $WCET$ (*Worst Case Execution Time*): Se trata del tiempo que se necesita para completar la tarea en el peor caso.
- Límite de tiempo D (*Deadline*): Momento temporal posterior a la activación de la tarea y antes del cuál se debe haber completado.
- Período T (*Period*): Es el tiempo que transcurre entre activaciones.

Estas tareas son ejecutadas por un reloj interno del sistema, el cuál debe ser lo suficientemente preciso como para asegurar que los ciclos se cumplen. Por otra parte, las tareas también pueden ser aperiódicas si son activadas por un evento externo (p. ej., un cambio en el entorno, como sucede en algunos sistemas de control) mediante mecanismos como las interrupciones. Las tareas aperiódicas siguen teniendo $WCET$ y D, pero no tienen período T. En algunos casos, existe un tiempo mínimo entre activaciones T, considerándose la tarea como esporádica. En ocasiones, los sistemas empotrados limitan su arquitectura para que solo haya tareas activadas por tiempo, evitando las interacciones y comprobando de forma periódica si se ha producido un evento. De todos estos atributos, el tiempo de ejecución en el peor caso es el más difícil de determinar. Se trata de un límite superior que puede ser estimado usando métodos estáticos, probabilísticos o, incluso, basados en mediciones. No obstante, la fiabilidad de muchos de estos métodos se apoya en supuestos sobre el sistema y su funcionamiento [45].

Para evaluar la efectividad de un algoritmo de planificación, solemos basarnos en su capacidad para cumplir con todas las restricciones temporales de un conjunto de tareas dado. Cuando se consiguen respetar todas estas restricciones, se dice que la planificación (o el conjunto de tareas) es viable, mientras que si el sistema se salta alguna, se suele decir que está sobrecargado. Un concepto muy importante en el que se apoyan los algoritmos de planificación para determinar la viabilidad de un conjunto de tareas es la utilización total de CPU para dicho conjunto, definida como

$$U = \sum_{i=1}^{n} \frac{C_i}{min(D_i, T_i)} \tag{2.1}$$

por Liu y Layland [46], donde C_i, D_i y T_i son el tiempo de ejecución, límite temporal y período para la tarea i del conjunto, respectivamente.

Anteriormente, se ha indicado que la planificación de procesos de tiempo real es apropiativa, de forma que el sistema operativo puede retirarle a un proceso los recursos que tenía asignado para dárselos a otro de mayor prioridad. La asignación de estas prioridades a las distintas tareas es la base de los algoritmos de planificación. Estos algoritmos pueden ser de dos tipos: estáticos o dinámicos. Los algoritmos estáticos realizan la planificación y asignan las prioridades a las tareas antes incluso de tener que poner en marcha el sistema, de forma *offline*, ejecutándose luego las tareas en el orden determinado por las prioridades establecidas. También conocida como planificación de prioridades fijas FPS (*Fixed-Priority Scheduling*) [47], para aplicar estos algoritmos es necesario conocer todas las características del problema de planificación (número de tareas y sus atributos) de antemano, lo cuál es imposible en algunas situaciones en las que se desconoce la carga de trabajo futura del sistema. Uno de los algoritmos estáticos más conocidos es RMS (*Rate Monotonic Scheduling*) [46][48], cuya idea principal es que las tareas con los períodos más pequeños tienen mayor prioridad. En la propuesta original de RMS realizada por Liu y Layland, asumen un conjunto de tareas periódicas con $D = T$ y determinan el límite superior para la utilización como

$$U \leq n(2^{\frac{1}{n}} - 1) \tag{2.2}$$

Esto implica que, para conjuntos de tareas muy grandes ($n \to \infty$), la utilización de CPU tiene que ser inferior al 70 %, que es una cifra relativamente baja. No obstante, si se cumplen todas estas condiciones, RMS garantiza encontrar una planificación viable si es que existe, o lo que es lo mismo, es un algoritmo óptimo. Otro problema de RMS es que requiere que el período sea igual al límite de tiempo para cada tarea. Para solucionar este problema, se propone DMT (*Deadline Monotonic Scheduling*) [49], en el que los procesos se caracterizan con $C_i \leq D_i \leq T_i$.

Como ya se ha comentado, el principal problema de la planificación estática es que se trata de algoritmos clarividentes, es decir, necesitan conocer toda la información sobre el conjunto de tareas para poder determinar su viabilidad. Los algoritmos dinámicos, por otra parte, permiten determinar esta viabilidad en tiempo de ejecución, cambiando las prioridades de las tareas sobre la marcha u *online*. Uno de los algoritmos dinámicos más conocidos es EDF (*Earliest Deadline First*) [46], donde las tareas con menor tiempo restante antes de su límite temporal son las que reciben una mayor prioridad. En este caso, el algoritmo es óptimo para conjuntos de tareas donde se cumpla la condición [50]

$$U \leq 1 \tag{2.3}$$

No obstante, esta situación se da solo en los sistemas uniprocesador. En los multiprocesador, determinar que la utilización es inferior al número de CPU no es una condición suficiente para garantizar que el conjunto de tareas tenga una planificación viable. En 2005, Anderson adapta EDF para sistemas multiprocesador [42], consiguiendo una condición suficiente para tareas de tiempo real blandas. Un enfoque diferente al de EDF lo proporciona MUF (*Maximum Urgency First*) [51]. En este algoritmo, se trabaja en torno a la urgencia de cada tarea, que es determinada según la combinación de varias prioridades. En concreto, cada tarea posee 2 prioridades fijas definidas antes de la ejecución y una dinámica, que se calcula sobre la marcha, existiendo una relación de precedencia entre estas prioridades.

En el caso de ser necesario trabajar con tareas aperiódicas o esporádicas, se puede aplicar alguno de los algoritmos propuestos en [52], entre los que destacamos DSA (*Deferrable Server Algorithm*) y SSA (*Sporadic Server Algorithm*).

Para terminar con el repaso de la computación de tiempo real, vamos a hacer un breve repaso de algunos de los RTOS más utilizados. En [53], Hambarde realiza una comparación entre Windows CE, VxWorks, QNX Neutrino y RTAI. El último, que es esencialmente una extensión del kernel de Linux o co-kernel que da soporte a este tipo de cargas de trabajo, obtiene los mejores resultados en latencia, *jitter* y tiempo de respuesta de entre los cuatro, si bien es cierto que su latencia ante interrupciones es mayor que la de VxWorks. Además de RTAI, otros co-kernels para Linux serían RTLinux y Xenomai. Todos estos co-kernels funcionan de manera similar: implementan un despachador de interrupciones que captura las interrupciones provenientes del hardware para redirigirlas a los procesos (tanto normales como de tiempo real), mientras que también poseen un planificador de procesos para asegurar que se cumplen las prioridades.

Además de los co-kernels, otra solución alternativa para Linux la encontramos en el parche `PREEMPT_RT`. El desarrollo de este parche es llevado a cabo por programadores del kernel dentro de la fundación Linux y en paralelo con el del propio kernel, lo que le da un cierto aire de «oficialidad». En esencia, los cambios que aplica este parche sobre el kernel habilitan la planificación apropiativa y aumentan el determinismo de ciertas operaciones. Este acercamiento, al trabajar sobre un único kernel en vez de los dos que se obtienen con las soluciones basadas en co-kernels, resulta en una mayor facilidad a la hora de implementar procesos de tiempo real, ya que se realiza de manera idéntica a los normales. Los co-kernels exponen interfaces especiales que se salen de la interfaz estándar de Linux y con las que sus procesos deben trabajar, algo que no ocurre con el parche `PREEMPT_RT` y que hace que sea muy fácil adaptar un proceso existente para que tenga un comportamiento más determinista. Este parche habilita varios modos de planificación de procesos nuevos para el planificador del kernel, entre los que destacaremos `SCHED_DEADLINE` [54]. Esta política de planificación aplica el algoritmo EDF junto con CBS (*Constant Bandwith Server*). El algoritmo CBS se encarga de asignar un ancho de banda a cada tarea, de forma que una tarea concreta no pueda ejecutarse durante un tiempo superior a su tiempo de ejecución dentro de cada período. De esta forma, se consigue el aislamiento temporal de las tareas, evitando que el comportamiento de una de ellas afecte a las demás. En cuanto a su rendimiento, el estudio llevado a cabo en [55] determina que no es una solución apta para sistemas de tiempo real duro, pero puede ser perfectamente usado cuando las restricciones son más livianas. En cualquier caso, el rendimiento va a depender enormemente del hardware utilizado y de la implementación de los drivers para dicho hardware.

Para este trabajo, se ha decidido usar como plataforma objetivo el sistema GNU/Linux parcheado con `PREEMPT_RT` debido a que se adhiere a la interfaz estándar del kernel y las tecnologías de virtualización mediante contenedores presentes en la plataforma son muy maduras. Además, las herramientas de prueba sobre esta plataforma también están muy asentadas. El hecho de que no sea un RTOS aceptable para sistemas con restricciones temporales duras nos es indiferente para el alcance de este proyecto, dado que solo se busca implementar una herramienta que sirva como prueba de concepto para el despliegue de tareas de tiempo real sobre múltiples dispositivos. En concreto, se usará la política de planificación `SCHED_DEADLINE` para las tareas desplegadas en dichos dispositivos, dado que la carga de trabajo que tendrá cada uno es desconocida (las tareas desplegadas sobre cada nodo pueden cambiar con el tiempo) y es la única planificación dinámica que ofrece Linux.

3. Análisis del rendimiento de procesos contenerizados

Para comprobar cómo de factible es la idea de la ejecución de procesos de tiempo real virtualizados mediante contenedores, hemos llevado a cabo una serie de pruebas con el objetivo de caracterizar algunos aspectos de su funcionamiento. Todas las pruebas han sido realizadas sobre el dispositivo Raspberry Pi detallado en la sección 1.4. Se ha hecho uso de la suite de pruebas `rt-tests`, la cuál contiene múltiples utilidades para obtener métricas sobre el rendimiento del kernel de Linux relevantes en entornos de tiempo real. Dentro de todas las pruebas posibles que contiene `rt-tests`, `cyclicdeadline` ha sido la utilizada. Esta prueba, derivada del conocido `cyclictest`, mide la latencia en la activación de tareas de tiempo real, es decir, la diferencia de tiempo entre el momento en el que se debería activar una tarea y en el que se activa realmente. La principal diferencia entre `cyclictest` y `cyclicdeadline` reside en que la primera mide la latencia para tareas planificadas con `SCHED_FIFO` (planificación estática), mientras que la segunda hace lo propio con `SCHED_DEADLINE`. Como se ha indicado en la sección 2.4, esta última es la política de planificación elegida para el sistema que se desea desarrollar, razón por la cuál hemos decidido usar `cyclicdeadline`.

Concretamente, se han hecho varias ejecuciones de esta prueba aumentando cada vez el número de hilos de tiempo real que se debían ejecutar y observar. Para todos los hilos, el límite de tiempo o *deadline* ha sido de $1000\mu s$. Estas pruebas se han realizado dos veces: primero, ejecutando `cyclicdeadline` de forma «nativa» y, luego, dentro de un contenedor. La definición de la imagen Docker que contiene la prueba, así como los pasos seguidos para su ejecución, son explicados detalladamente en el apéndice C. Los resultados obtenidos son los mostrados en la tabla 3.1. Cabe destacar que `cyclicdeadline` solo nos devuelve al final de su ejecución, para cada uno de los hilos, los valores mínimo, máximo y medio de la latencia. Por ello, los resultados de la tabla para ejecuciones con más de un hilo muestran realmente el mínimo de los mínimos de los hilos, el máximo de los máximos y la media de las medias.

Lo que queríamos observar con estas pruebas es el impacto que tiene sobre la latencia el uso de Docker. En la gráfica de la figura 3.1, se muestran los datos de la anterior tabla de forma que este impacto se hace evidente. Aunque la latencia media se mantiene relativamente similar, el límite superior es mayor en las ejecuciones contenerizadas. No obstante, el impacto no es especialmente notable, de forma que se puede seguir considerando como una herramienta factible para el despliegue de procesos de tiempo real, si bien no será recomendable para los que tengan unas restricciones temporales más pequeñas donde esta latencia sea demasiado alta.

Hay que tener en cuenta que la ejecución de `cyclicdeadline` dentro de un contenedor se realiza de forma que este programa lanza los hilos de tiempo real dentro de su mismo contenedor, el cuál ya se encuentra en ejecución. Por ello, se decidió realizar una segunda prueba con el objetivo de analizar la latencia en el lanzamiento de los propios contenedores, dato que es especialmente importante en el contexto que se plantea para la herramienta de orquestación

Tipo	Hilos	Latencia mínima (μs)	Latencia máxima (μs)	Latencia media (μs)
Contenedor	1	1	113	22.0
Contenedor	2	1	257	85.0
Contenedor	3	1	362	127.33
Contenedor	4	1	426	112.0
Contenedor	5	1	527	210.20
Contenedor	6	1	700	263.0
Nativo	1	1	88	17.0
Nativo	2	1	267	123.0
Nativo	3	1	332	149.0
Nativo	4	1	307	122.0
Nativo	5	3	448	196.0
Nativo	6	1	613	205.67

Tabla 3.1.: Resultados obtenidos de la ejecución de `cyclicdeadline`.

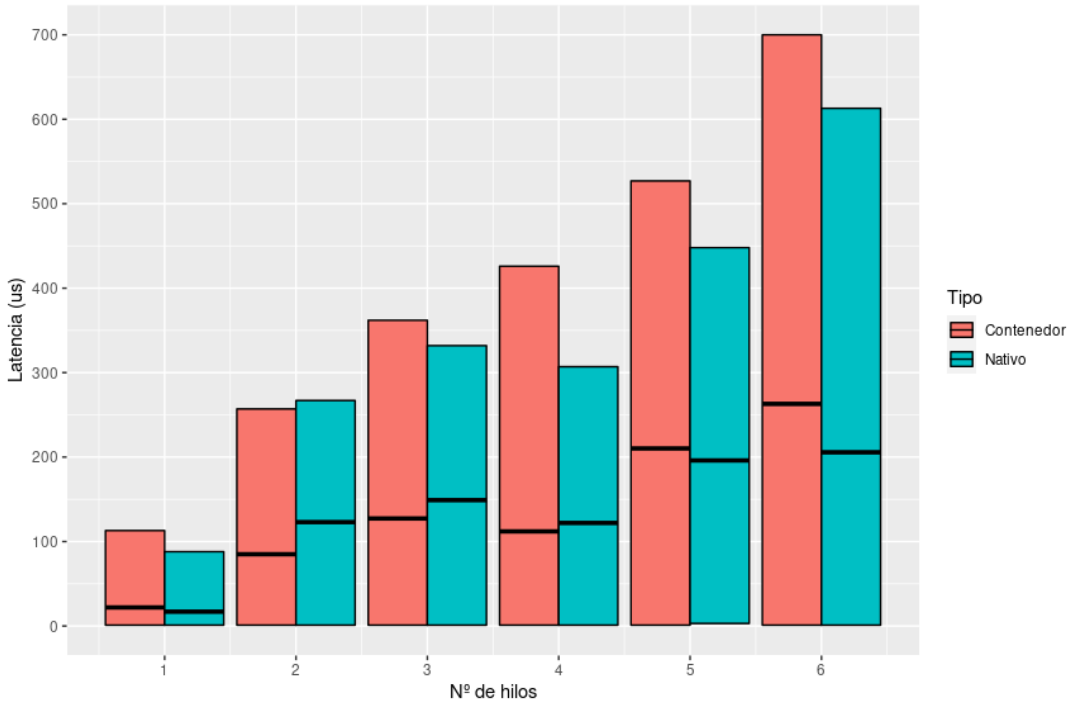


Figura 3.1.: Gráfica de intervalos con los rangos de latencia mínima a máxima obtenidos de la ejecución de `cyclicdeadline`. Para cada rango, se muestra también su media.

de tareas de tiempo real, dado que estas tareas se deben lanzar en los nodos desde cero. Para que la ejecución del proceso de prueba no influya en los resultados, se ejecuta todo este proceso desde un ordenador externo «maestro». El funcionamiento es el siguiente:

1. El maestro lanza un contenedor en la Raspberry Pi mediante SSH.

2. Cuando el contenedor se lanza, envía un mensaje UDP de vuelta al maestro informando de que ha terminado su ejecución.

3. El maestro obtiene los tiempos de creación e inicio del contenedor ejecutando el comando `docker inspect` mediante SSH.

4. La latencia se obtiene como la diferencia entre ambos tiempos.

5. El maestro elimina el contenedor de la Raspberry Pi mediante SSH para poder lanzarlo de nuevo en la siguiente iteración.

Para comprobar también si el lenguaje usado en la implementación del proceso contenerizado tiene algún impacto sobre esta latencia, se han usado tres versiones diferentes del contenedor que se lanza sobre la Raspberry Pi, usando C, Go y Rust. Los tres son lenguajes compilados con similitudes en cuanto a su alcance (programación de sistemas) y características. En total, se han realizado 1000 iteraciones de esta prueba para cada uno de los lenguajes. Los resultados obtenidos se muestran en la tabla 3.2. En la figura 3.2 se puede observar una representación gráfica de todas las latencias observadas para cada iteración.

Lenguaje	**Tiempo mínimo** (ms)	**Tiempo máximo** (ms)	**Tiempo medio** (ms)	**Desviación típica**
C	1472	2548	1693	165
Go	1445	2718	1693	163
Rust	1478	2378	1701	156

Tabla 3.2.: Valores observados para la latencia en el lanzamiento de contenedores implementados en distintos lenguajes.

A primera vista, apreciamos que el lenguaje usado para la implementación de los procesos contenerizados no tiene ningún impacto sobre la latencia en su lanzamiento. Por otro lado, se trata de valores muy altos, demasiado como para que podamos considerar el lanzamiento de un contenedor como parte de una tarea de control dura. También destaca la gran variación de estas latencias. Todo esto nos hace pensar que la implementación que tiene Docker de sus tareas de gestión del ciclo de vida de los contenedores no están preparadas para ser deterministas, aunque, obviamente, dudamos de que este aspecto fuera considerado en su diseño.

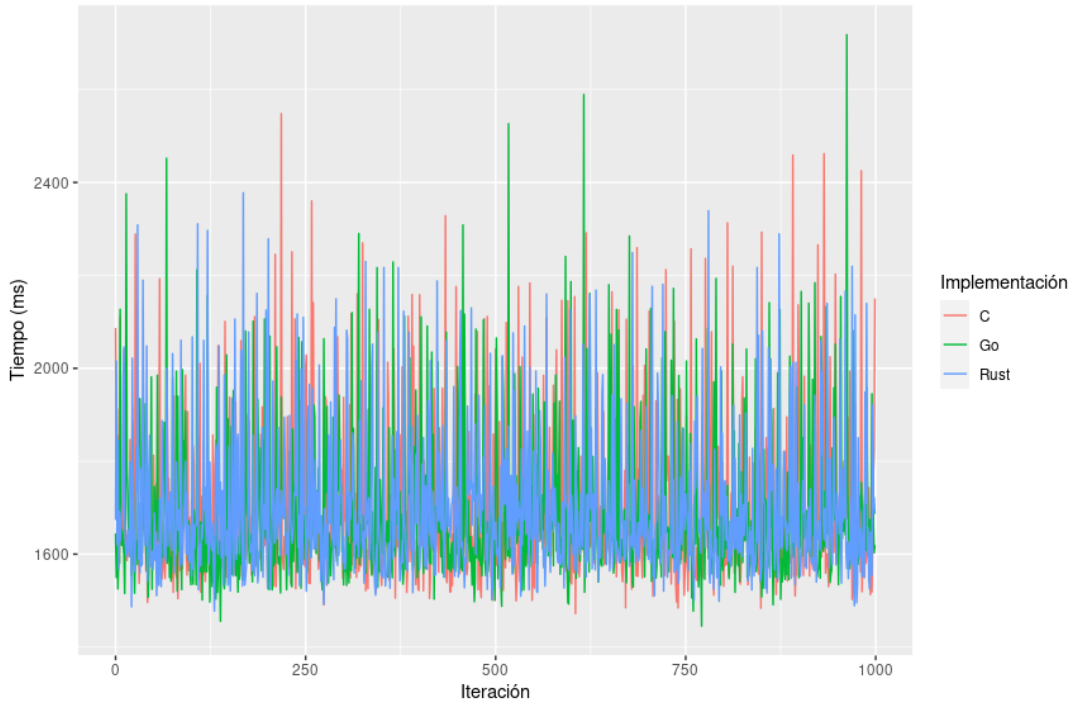


Figura 3.2.: Gráfico de líneas con las latencias observadas en el lanzamiento de contenedores implementados con distintos lenguajes.

4. Diseño e implementación de un orquestador para tareas de tiempo real

En los capítulos anteriores, se ha presentado el concepto de la Industria 4.0 y los problemas que plantea, indicando que el paradigma del *fog computing* y las tecnologías de virtualización pueden ser una solución factible para conseguir su implantación. Para apoyar este planteamiento, se ha decidido implementar una herramienta de orquestación de tareas de tiempo real sobre entornos distribuidos que sirva como prueba de concepto. En este capítulo, se detalla el proceso seguido para su desarrollo, justificando las decisiones de diseño tomadas y mostrando los de detalles más relevantes de la implementación. Para terminar, se muestra una caracterización inicial del rendimiento de la herramienta desarrollada.

4.1. Objetivos y requisitos

Como ya se ha explicado, la llegada de la Industria 4.0 supone un aumento muy grande en la cantidad de datos generados en las plantas debido al IoT industrial, datos que son necesarios para poder obtener gemelos digitales con un nivel de detalle suficiente. Los nuevos sistemas ciber-físicos necesitan, por tanto, procesar todos estos datos en tiempo real para poder tomar decisiones de control en base a los mismos y garantizar una operación eficiente. Para ello, es necesario poseer una capacidad de procesamiento elevada, mayor de la que cualquier dispositivo individual pueda aportar. Aunque la computación en la nube pueda parecer una solución natural a este problema, la enorme carga que pondría sobre la red la constante transmisión de cantidades de datos tan grandes, así como la latencia resultante de estas comunicaciones, hacen que no podamos considerar esta plataforma como apropiada para dar cobijo a tareas con restricciones temporales. En la sección 2.2, se introduce el paradigma *fog* como una extensión de la nube más cercana a las fuentes de datos. Según este modelo, los datos generados por los dispositivos del borde de la red son procesados en nodos ubicados en la misma red local. Aunque la potencia combinada de los dispositivos que conformen esta capa *fog* será menor que la que encontramos en la nube, puede ser suficiente para las tareas de análisis de datos y toma de decisiones que se deben realizar en las plantas, reduciendo así la presión sobre la red y garantizando unos tiempos de respuesta muy inferiores.

Debido a esto, el modelo *fog* se plantea como una posible plataforma para la implementación de los nuevos sistemas ciber-físicos. Las tareas de control industrial deben, entonces, distribuirse por los nodos de la capa *fog* de forma dinámica para dar respuesta a la carga de trabajo en todo momento. Para que esta distribución de procesos sobre los nodos se produzca de manera eficiente, es necesario hacer uso de las tecnologías de virtualización, las cuales ya sirven para solucionar una problemática similar en la nube. Al virtualizar los procesos, se abstraen las capas inferiores como son el hardware o el sistema operativo, facilitando el despliegue de los mismos sobre dispositivos con características dispares y unificando su desarrollo, dado que no

es necesario implementarlos para cada plataforma distinta). Así, se facilita la escalabilidad de los procesos para dar respuesta a los cambios en la demanda. En la sección 2.3, planteamos el uso de contenedores en vez de máquinas virtuales para esto, apoyándonos en su carácter más liviano y su mejor rendimiento para operaciones de entrada y salida.

Por tanto, en el modelo planteado es necesario desplegar las tareas de control en forma de contenedores. Para explorar más este concepto, hemos decidido implementar una herramienta que permita realizar esto sobre múltiples nodos. Los objetivos de esta herramienta son:

Nombre	Gestión de nodos
Importancia	Alta
Descripción	Llevar un control de los dispositivos que conforman la capa *fog* es una parte esencial de la implantación de este modelo.

Tabla 4.1.: Objetivo 01 - Gestión de nodos.

Nombre	Gestión de tareas
Importancia	Alta
Descripción	Las distintas tareas que se deben ejecutar sobre los nodos de la capa *fog* deben estar recopiladas y centralizadas en el sistema.

Tabla 4.2.: Objetivo 02 - Gestión de tareas.

Nombre	Orquestación de tareas
Importancia	Muy alta
Descripción	El sistema debe permitir a los usuarios desplegar las tareas definidas sobre los nodos de manera sencilla, controlando siempre que la ejecución de las tareas sobre cada nodo concreto sea viable.

Tabla 4.3.: Objetivo 03 - Orquestación de tareas.

Nombre	Uso de software libre
Importancia	Media
Descripción	Como parte de nuestro compromiso con el software libre, el sistema desarrollador deberá hacer uso siempre que sea posible de tecnologías abiertas.

Tabla 4.4.: Objetivo 04 - Uso de software libre.

A partir de estos objetivos, se han concretado una serie de requisitos que deberá satisfacer la herramienta desarrollada.

Nombre	Añadir un nuevo nodo
Objetivos relacionados	4.1
Descripción	Un usuario debe poder añadir al sistema un nuevo nodo que represente a un dispositivo de la capa *fog*. Los datos proporcionados para el nuevo nodo deben permitir la conexión al mismo por SSH.

Tabla 4.5.: Requisito 01 - Añadir un nuevo nodo.

Nombre	Obtener datos de los nodos
Objetivos relacionados	4.1
Descripción	Un usuario debe poder obtener información sobre los nodos que hay registrados en el sistema, tanto de forma colectiva como detallada para un nodo concreto.

Tabla 4.6.: Requisito 02 - Obtener datos de los nodos.

Nombre	Modificar un nodo
Objetivos relacionados	4.1
Descripción	Un usuario debe poder actualizar la información relativa a un nodo ya registrado en el sistema si fuera necesario.

Tabla 4.7.: Requisito 03 - Modificar un nodo.

Nombre	Eliminar un nodo
Objetivos relacionados	4.1
Descripción	Un usuario debe poder eliminar del sistema un nodo cuando sea necesario.

Tabla 4.8.: Requisito 04 - Eliminar un nodo.

Nombre	Añadir una nueva tarea
Objetivos relacionados	4.2
Descripción	Un usuario debe poder añadir al sistema tareas de tiempo real para su posterior despliegue sobre los nodos del mismo. Se deben aportar, por tanto, tanto los ficheros con la tarea como los atributos relativos a sus restricciones temporales.

Tabla 4.9.: Requisito 05 - Añadir una nueva tarea.

Nombre	Obtener datos de las tareas
Objetivos relacionados	4.2
Descripción	Un usuario debe poder ver la información relativa a las tareas registradas en el sistema.

Tabla 4.10.: Requisito 06 - Obtener datos de las tareas.

Nombre	Modificar una tarea
Objetivos relacionados	4.2
Descripción	Un usuario debe poder actualizar una tarea ya presente en el sistema si fuera necesario.

Tabla 4.11.: Requisito 07 - Modificar una tarea.

Nombre	Eliminar una tarea
Objetivos relacionados	4.2
Descripción	Un usuario debe poder eliminar una tarea del sistema.

Tabla 4.12.: Requisito 08 - Eliminar una tarea.

Nombre	Desplegar una tarea en un nodo
Objetivos relacionados	4.3
Descripción	Un usuario debe poder desplegar una tarea registrada en el sistema sobre un nodo también registrado.

Tabla 4.13.: Requisito 09 - Desplegar una tarea en un nodo.

Nombre	Eliminar una tarea de un nodo
Objetivos relacionados	4.3
Descripción	Un usuario debe poder eliminar una tarea previamente desplegada en un nodo cuando sea necesario.

Tabla 4.14.: Requisito 10 - Eliminar una tarea de un nodo.

Nombre	Garantizar la viabilidad de los conjuntos de tareas
Objetivos relacionados	4.3
Descripción	El sistema debe de garantizar en todo momento y en la medida de lo posible que las tareas que se ejecutan sobre un nodo dado son viables.

Tabla 4.15.: Requisito 11 - Garantizar la viabilidad de los conjuntos de tareas.

Nombre	Despliegue usando contenedores
Objetivos relacionados	4.3
Descripción	El despliegue de las tareas sobre los nodos se debe realizar mediante contenedores.

Tabla 4.16.: Requisito 12 - Despliegue usando contenedores.

4.2. Diseño del sistema

A la hora de abordar el diseño de la herramienta, se nos planteaban varias posibilidades. La solución más sencilla podría consistir de una aplicación monolítica, de escritorio o web, que aglutinase toda la funcionalidad recogida en los requisitos detallados en la sección anterior. Una aplicación monolítica de este tipo, aunque no siempre es una mala opción, plantea varios problemas como son su difícil escalabilidad o la complejidad creciente conforme se añaden funcionalidades, lo que incrementa los costes de desarrollo y mantenimiento. Además, el sistema que se desea crear se visualiza siendo usado por varios usuarios. Se podría pensar, por ejemplo, en ingenieros de una fábrica que se encargan del desarrollo y mantenimiento de sus sistemas de control, cada uno usando su propio PC para llevar a cabo estas tareas a través de nuestra herramienta. En una situación como esa, usar una aplicación monolítica es inviable, dado que los usuarios deben trabajar de forma concurrente sobre un conjunto de datos común.

La manera más sencilla para permitir esta colaboración y conseguir que todos los usuarios trabajen sobre una misma base consiste en aplicar un modelo de cliente-servidor. El sistema se modulariza, encapsulando la funcionalidad más importante en un servidor y relegando las aplicaciones que usan los usuarios a simples cascarones encargados de proporcionar una interfaz entre las funcionalidades del sistema y ellos mismos. Una arquitectura de este tipo hace más fácil controlar el acceso concurrente a los datos, evitando problemas de integridad e incoherencia en los mismos. Además, debido a que los clientes no tienen tantas responsabilidades, acaban siendo más simples y livianos, lo que facilita su instalación en las máquinas de los usuarios. Es por todas estas razones que se ha escogido la arquitectura cliente-servidor para el diseño del sistema. En la figura 4.1 se puede ver un diagrama del diseño propuesto para el sistema. Aquí se puede ver como el servidor contiene toda la lógica relativa a los nodos, las tareas y el despliegue de éstas. Aunque se podría haber planteado la separación de esta lógica en varios microservicios, esta opción se descartó debido a que no se espera que el sistema tenga que ser usado por muchos usuarios a la vez ni su carga pueda elevarse de forma considerable, por lo que los beneficios de escalabilidad que este tipo de diseño podría aportar son innecesarios y se ven enormemente contrarrestados por la complejidad que añaden.

Volviendo a la figura 4.1, apreciamos que el servidor también controla el acceso a la base de datos donde se almacena toda la información sobre los nodos y las tareas. Para esta base de datos, como ya se indicó en la sección 1.4, se ha decidido MongoDB, que es un gestor de bases de datos documentales y no estructuradas. También se puede comprobar en el diagrama que el servidor expone todas las funcionalidades del sistema a través de una API (*Application Programming Interface*). Esta API sigue el patrón arquitectónico REST (*REpresentation State Transfer*), el cuál fue introducido en el año 2000 por Roy Fielding en su tesis doctoral [56]. Simplificando los conceptos que se desarrollan en esta tesis, podemos identificar las siguientes características fundamentales del patrón REST:

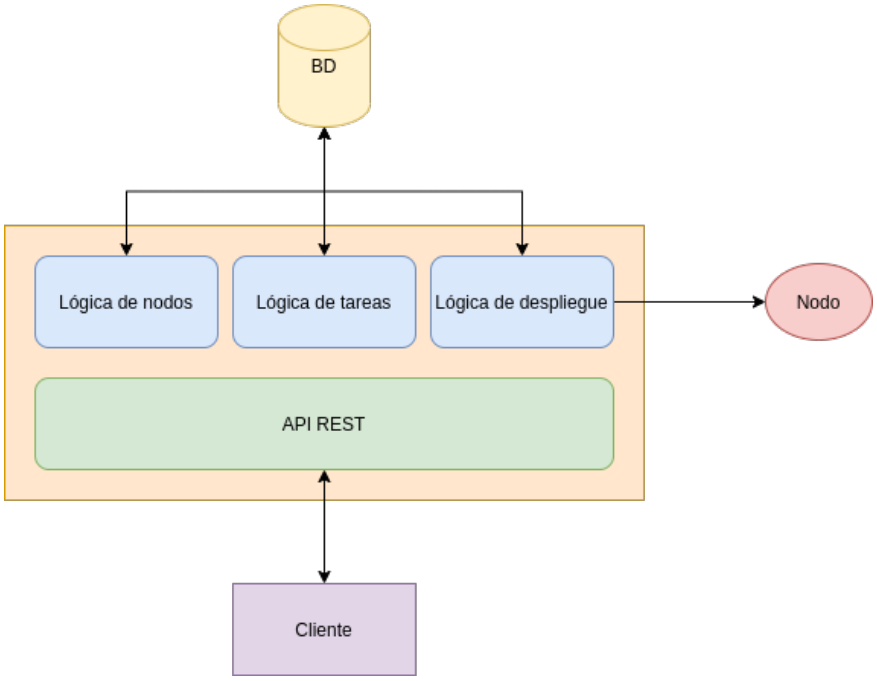


Figura 4.1.: Diagrama de arquitectura del sistema.

- La interfaz se diseña en torno a recursos, los cuáles son conjuntos de información.
- Cada recurso está identificado de forma única mediante una URI.
- Las operaciones se realizan mediante peticiones HTTP hacia la URI del recurso objetivo.
- La operación a realizar viene definida por el verbo HTTP usado en la petición (`GET`, `POST`, `PUT`, `PATCH` o `DELETE`).
- El servidor no almacena estado alguno, de forma que cada petición debe contener toda la información necesaria como para poder darle respuesta.
- Para la transferencia de información, se utilizan formatos bien definidos como JSON o XML.

La aplicación del patrón REST al diseño de interfaces hace que sea más sencillo estructurar el acceso a la información que contiene un servicio concreto, además de simplificar también el flujo de trabajo entre cliente y servidor. Para aplicar REST al diseño de un servicio concreto, se debe identificar cuáles son los recursos con los que trabaja y qué operaciones permite sobre los mismos. En el caso de nuestro sistema, tenemos los recursos detallados en las tablas 4.17 y 4.18, mientras que las operaciones posibles se presentan en la sección 4.3.1.

Recurso	Nodo
Atributos	▪ ID ▪ Nombre ▪ Dirección IP ▪ Usuario (utilizado para la conexión SSH) ▪ CPU (modelo del procesador) ▪ Arquitectura de CPU (p. ej., `ARMv7`) ▪ Número de núcleos de la CPU ▪ Frecuencia de reloj ▪ RAM ▪ Dispositivos (p. ej., `/dev/null`) ▪ Tareas (que se están ejecutando en el nodo)

Tabla 4.17.: Definición del recurso nodo.

Recurso	Tarea
Atributos	▪ ID ▪ Nombre ▪ Tiempo de ejecución (*runtime*) ▪ Límite de tiempo (*deadline*) ▪ Período (*period*) ▪ Dispositivos (a los que necesita acceder) ▪ Capacidades (p. ej., `SYS_NICE`) ▪ ID del desplegable (fichero tar con código fuente y `Dockerfile`)

Tabla 4.18.: Definición del recurso tarea.

Como ya se ha indicado, aplicar una arquitectura cliente-servidor como esta permite reducir al máximo la lógica que contienen los clientes, los cuáles solo tienen que ocuparse de mostrar la interfaz, comprobar las entradas de los usuarios y realizar la comunicación con el servidor. Además, el hecho de que el servidor exponga una API estandarizada facilita enormemente la implementación de múltiples tipos de aplicaciones cliente, incluso para distintas plataformas

(p. ej., móvil, web o escritorio). Estos clientes sencillamente tienen que consumir recursos de la interfaz y pueden ofrecer a sus usuarios exactamente las mismas funcionalidades que los demás. Esta separación de responsabilidades facilita también el desarrollo y mantenimiento del software, ya que se trata de partes completamente independientes y separadas que solo deben atenerse a la especificación de la interfaz mediante la que se conectan.

Por último, también se ha decidido en esta fase de diseño implementar una imagen Docker, que es el motor de contenedores usado por el sistema, para facilitar el despliegue de tareas de tiempo real usando el sistema planteado. El funcionamiento de esta imagen, así como los detalles de diseño e implementación de las demás partes del sistema, se explican de forma extensiva en las siguientes secciones de este capítulo. Todo el código desarrollado del que se va a hablar se encuentra disponible en los repositorios de GitHub del proyecto[1][2][3].

4.3. Servidor maestro

Tal y como se ha descrito en la sección anterior, el servidor es la parte del sistema desarrollado que encapsula la lógica principal encargada de llevar a cabo las funcionalidades deseadas. En esta sección se ahonda en las decisiones de diseño tomadas para este componente, además de mostrar algunas partes interesantes de la implementación y las técnicas de prueba e integración continua aplicadas.

4.3.1. Diseño e implementación

A la hora de estructurar el servidor, decidimos hacer una separación por dominios, de forma que tenemos por un lado la lógica relativa a las funcionalidades de los nodos y, por otro, la de las tareas. A su vez, también realizamos una división de responsabilidades, diferenciando entre controladores y servicios.

Los controladores definen las rutas y operaciones posibles en la API REST del servidor. Este servidor recibe peticiones HTTP para las rutas que expone, las cuáles se manejan con distintos controladores dependiendo del verbo HTTP usado. Estos controladores se encargan entonces de validar los datos recibidos en la petición, gestionar la autenticación y construir las respuestas con el código de estado adecuado, llamando a funciones de los servicios para realizar las operaciones. En los servicios se implementa la lógica encargada de llevar a cabo estas operaciones como tal, conectando con la base de datos para la gestión de nodos y tareas, además de llevar a cabo el despliegue de éstas. Como ya se ha indicado, la lógica se divide también por dominio, de forma que tenemos unos controladores para las operaciones sobre los nodos y otros separados para las que ocurren sobre las tareas. Lo mismo ocurre con la lógica de estas operaciones en los servicios. Para las acciones de orquestación de tareas, además, los servicios hacen uso de un paquete independiente que implementa estas funcionalidades concretas. El diagrama de la figura 4.2 refleja esta arquitectura que se acaba de describir.

[1] Repositorio del servidor: `https://github.com/Varrrro/shipyard-server`
[2] Repositorio del cliente: `https://github.com/Varrrro/shipyard-cli`
[3] Repositorio de la librería: `https://github.com/Varrrro/libshipyard`

En cuanto a las operaciones concretas que se pueden realizar mediante la API REST del servidor, se han definido e implementado las siguientes:

- `GET /nodes`

  Con esta operación, se obtiene una lista completa de los nodos presentes en el sistema. En caso de ocurrir algún error (p. ej., en la conexión con la base de datos), el servidor devuelve un una respuesta 500 (*Internal Server Error*). Si no hay ningún problema, la respuesta tiene el código de estado 200 (*Ok*) y contiene en su cuerpo la lista de nodos. Cabe destacar que en esta lista de nodos no se muestra toda la información presente en la base de datos sobre los mismos, sino que se ocultan algunos datos como son el usuario SSH, los dispositivos que tiene o los detalles de la CPU.

- `POST /nodes`

  Esta operación permite crear un nuevo nodo, debiendo incluir en el cuerpo de la petición los datos del nuevo nodo. No es necesario proporcionar valores para todos los atributos definidos en la tabla 4.17, sino que solo son requeridos el nombre, dirección IP y número de núcleos de la CPU del nuevo nodo. El nombre dado debe ser único, devolviendo el servidor una respuesta 409 (*Conflict*) si ya está en uso por otro nodo del sistema.

  La petición HTTP que recibe el servidor debe contener en la cabecera *Authorization* unas credenciales de autenticación de tipo básico, es decir, usuario y contraseña codificados con Base64. Este usuario y contraseña son los usados por el servidor para establecer la conexión SSH inicial con el nodo. Esta conexión inicial sirve para añadir los datos del nodo al fichero `known_hosts` del servidor con el que se regulan las conexiones SSH conocidas y copiar la clave pública de éste al fichero `authorized_keys` del nodo. Esto permite que las siguientes conexiones SSH que deba realizar el servidor usen esta clave para la autenticación, de forma que no sea necesario usar la contraseña y aumentando la seguridad.

  Si los datos enviados en el cuerpo de la petición no son correctos, el controlador devuelve una respuesta 400 (*Bad Request*). En caso de no poder crear el nodo por cualquier otro motivo, se devuelve un código 500. La ejecución correcta de esta operación produce una respuesta con código de estado 200 y con el nuevo recurso recién creado en el cuerpo.

- `GET /nodes?name=<node_name>`

  Como se puede suponer por el verbo HTTP y la ruta usadas, esta operación permite obtener la información detallada de un nodo concreto, el cuál se identifica por el nombre proporcionado. Esta operación y `GET /nodes` comparten controlador, comprobándose al inicio de éste si en la URI de la petición se encuentra el parámetro de consulta *name* para decidir si se llama a la función del servicio de nodos que devuelve la lista o a la que devuelve los detalles de uno concreto. A diferencia de lo que ocurría con esta otra operación, en este caso sí que se devuelven todos los atributos del nodo indicado junto con la respuesta 200. Si no se encuentra en la base de datos ningún nodo con el nombre indicado, la respuesta devuelta tiene un código de estado 404 (*Not Found*).

- `GET /nodes/<node_id>`

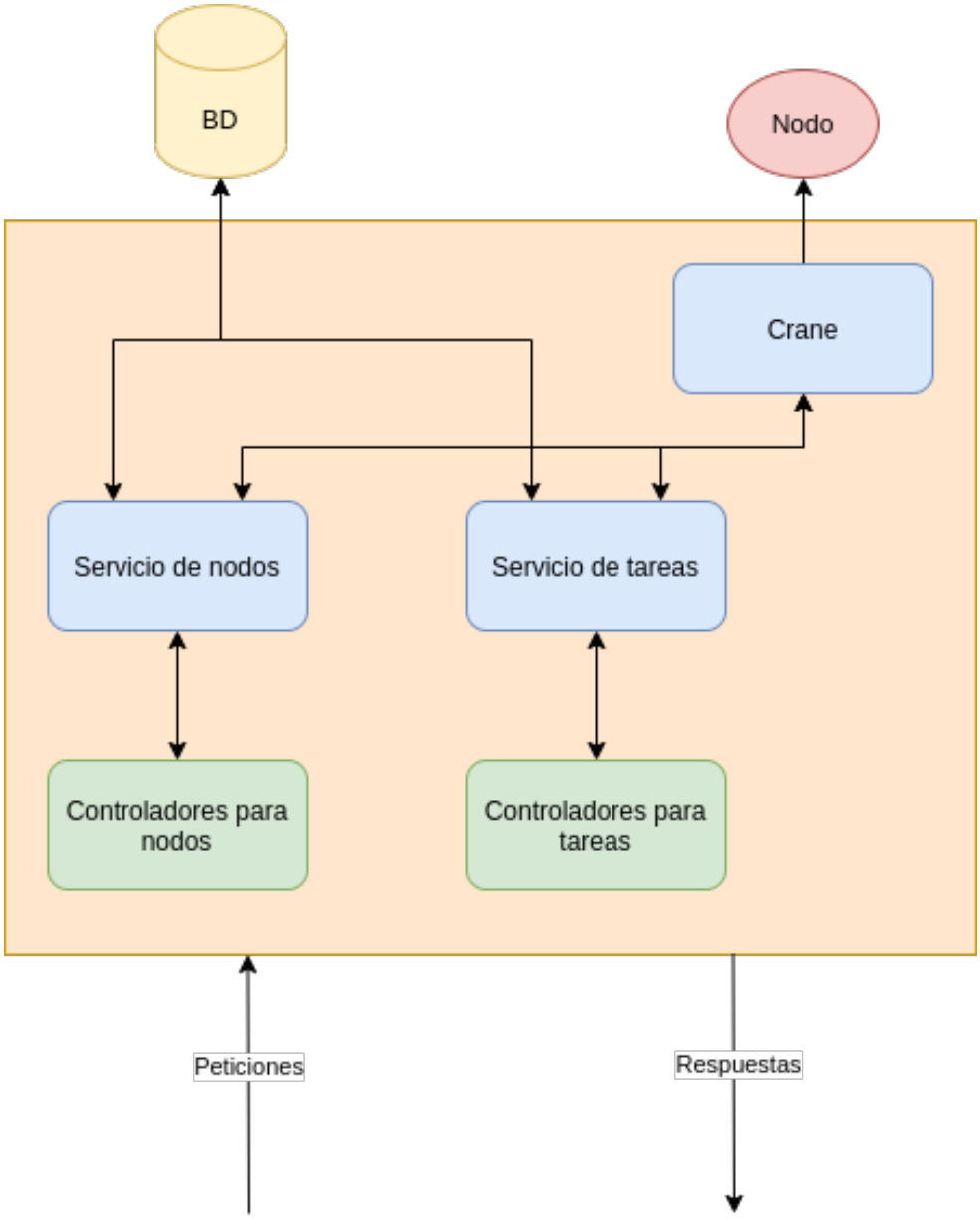


Figura 4.2.: Diagrama de arquitectura del servidor.

Esta operación es idéntica a la anterior, siendo la única diferencia que la identificación del nodo cuyos detalles se desean obtener se realiza por su ID. Esta operación también devuelve una respuesta 200 con los datos del nodo en el cuerpo en caso de encontrarse en la base de datos y un código 404 en caso contrario. También puede devolver una raspuesta 409 si el ID dado no es válido, es decir, si no cumple con el formato de ID de MongoDB.

- `PUT /nodes/<node_id>`

  Con esta operación se pueden modificar los atributos de un nodo ya existente en el sistema. En el cuerpo de la petición se incluyen solo los atributos cuyo valor se desea cambiar junto con estos nuevos valores. Si el formato del cuerpo de la petición no es correcto, el controlador de la ruta devuelve una respuesta 409. También se devuelve este código si el ID indicado no es válido. Si no se encuentra en la base de datos del sistema ningún nodo con el ID dado, se devuelve una respuesta 404. En caso de realizarse correctamente la operación, la respuesta tiene el código de estado 200 y contiene en su cuerpo todos los datos del recurso actualizado. Cabe destacar que, si alguno de los atributos modificados es el nombre, la IP, el usuario SSH o la lista de dispositivos del nodo, el servidor procede a eliminar del mismo todas las tareas que tuviera ejecutándose con el fin de evitar inconsistencias y estados no recuperables en el sistema. En el diagrama de la figura 4.3 se refleja este comportamiento, además del resto de acciones que lleva a cabo el servidor para realizar esta operación.

- `DELETE /nodes/<node_id>`

  Esta operación permite eliminar un nodo del sistema. Las peticiones no necesitan contener ningún dato en el cuerpo, siendo solo necesario indicar en la propia ruta de la petición el ID del nodo a eliminar. Si este ID no es válido, el controlador devuelve una respuesta 409, mientras que si no se encuentra ningún nodo con este ID la respuesta es 404. Si la eliminación del recurso se realiza correctamente, la respuesta tiene el código de estado 200 y contiene en su cuerpo los datos del recurso recién eliminado. Al igual que ocurría con la modificación, el sistema se asegura de parar todas las tareas que se estuvieran ejecutando sobre el nodo que se elimina y borrarlas del mismo, de forma que no queden «restos» en el dispostivo.

- `GET /tasks`

  De forma análoga a la operación `GET /nodes`, ésta devuelve la lista de tareas que están almacenadas en la base de datos del sistema. El cuerpo de la respuesta contiene esta lista con el código de estado 200 si la operación se ha podido realizar correctamente, devolviendo el código 500 en caso contrario.

- `POST /tasks`

  Permite añadir una nueva tarea al sistema. A diferencia de las demás operaciones, la cabecera *content-type* de las peticiones para esta operación, que es la que regula el tipo de contenido, no tiene el valor `application/json`, sino que se usa `multipart/form-data`. Este tipo de contenido se compone de múltiples campos, los cuáles pueden ser textuales o archivos, lo cuál es necesario para esta operación dado que es necesario aportar tanto los metadatos de la tarea en formato JSON como un fichero `tar.gz` con el código fuente

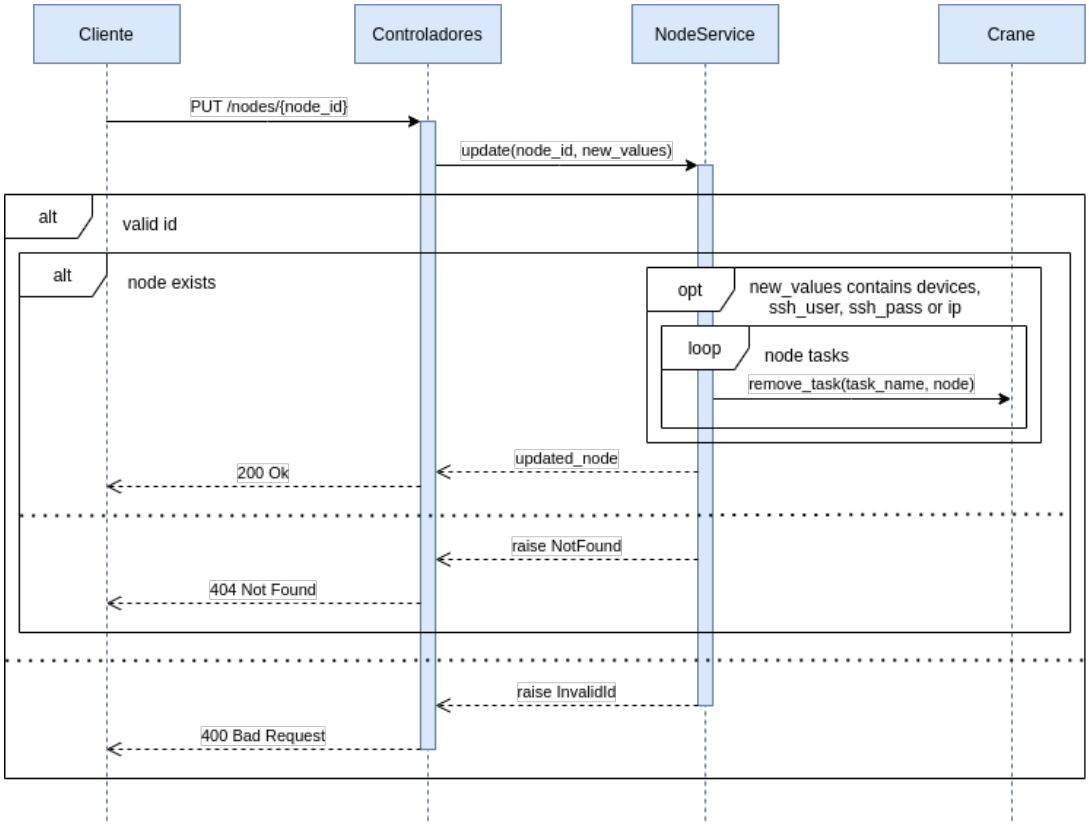


Figura 4.3.: Diagrama de secuencias de la operación de actualización de un nodo.

y la definición de la imagen Docker. El controlador de esta operación se encarga de extraer todos estos datos de los dos campos del cuerpo de la petición y, entonces, llamar a la función del servicio encargada de realizar la inserción de la tarea en la base de datos, devolviendo el controlador una respuesta 200 si la operación se realiza correctamente. Si el formato de la petición recibida es incorrecto (p. ej., falta alguno de los campos), se devuelve una respuesta 400. Además, el nombre de la nueva tarea debe ser único, obteniendo una respuesta 409 en caso de no serlo. Si no se puede crear la tarea por cualquier otra razón, la respuesta obtenida es 500.

- `GET /tasks?name=<task_name>`

  Con esta operación se pueden obtener los detalles de una tarea concreta, la cuál se identifica a partir del nombre indicado en el parámetro de consulta *name* de la petición. Al igual que ocurría con la operación `GET /nodes?name=<node_name>`, el controlador que da respuesta a esta operación es compartido con el de `GET /tasks`. Al inicio de la ejecución, el controlador comprueba la presencia del parámetro *name* en la petición, devolviendo los detalles de la tarea o la lista completa de tareas según si está presente o no. En caso de operación correcta, la respuesta 200 incluye en el cuerpo los datos del recurso deseado a excepción del fichero `tar.gz` con su código fuente. Si este recurso no se encuentra, el código devuelto es 404, devolviendo 500 en cualquier otro caso.

- `GET /tasks/<task_id>`

  Al igual que la operación anterior, ésta también devuelve los detalles de una tarea, aunque en este caso se usa el ID de la misma para su identificación. Si este ID no es válido, el controlador devuelve una respuesta 400; mientras que si no se encuentra en la base de datos ninguna tarea con este ID, la respuesta es 404. Esta operación tampoco devuelve el fichero de despliegue asociado a la tarea. Si no se puede realizar la operación, el controlador devuelve una respuesta 500.

- `PUT /tasks/<task_id>`

  Modificación de una tarea. Como ocurría en la operación de creación de tareas, las peticiones también tienen el tipo de contenido `multipart/form-data`. Como diferencia, no es necesario que la petición contenga tanto el dichero `tar.gz` con el código fuente como el JSON con los atributos, sino que solo debe contener lo que se vaya a actualizar. Antes de modificar la tarea, el servidor se asegura de eliminarla de cualquier nodo en el que se estuviera ejecutando para evitar inconsistencias en el sistema. Si el ID dado o el formato del cuerpo de la petición no son válidos, el controlador devuelve una respuesta 400. Si no se encuentra en la base de datos ninguna tarea con el ID indicado, se devuelve un 404. En caso de realizarse la operación correctamente, la respuesta devuelta tiene el código 200 y contiene en su cuerpo el recurso actualizado. Si no se puede realizar la operación por cualquier otra razón, devuelve una respuesta 500.

- `DELETE /tasks/<task_id>`

  Elimina una tarea de la base de datos del sistema, identificada por el ID indicado en la propia petición. Si no se encuentra ninguna tarea con dicho ID, devuelve 404. Si este ID no es válido, devuelve 400. Si no se puede realizar la operación por cualquier otra razón, devuelve 500. Cuando la eliminación se completa de forma satisfactoria, el controlador

devuelve una respuesta con código 200 y con el recurso recién eliminado en el cuerpo.

- `POST /nodes/<node_id>/tasks?task_id=<task_id>`

  Esta operación permite el despliegue de una tarea dada en un nodo. Es, quizás, la operación más compleja que realiza el servidor. Se recibe una petición en cuya ruta se se encuentra el ID del nodo, aportándose el de la tarea como un parámetro de consulta (*query*). Si este parámetro de consulta no está presente en la petición, el controlador devuelve inmediatamente una respuesta 400. Si todos los valores están presentes, se llama entonces a la función del servicio encargada de llevar a cabo el despliegue.

  Esta función convierte los IDs a los propios de MongoDB, lanzando un error `InvalidId` en caso de que su formato no sea correcto. El controlador devolvería entonces una respuesta 400 al cliente. El servicio puede devolver un error `NotFound` si no encuentra en la base de datos los recursos deseados, error que captura el controlador para generar una respuesta 404. Si los IDs se corresponden con recursos existentes en el sistema, el servicio procede a comprobar que el nodo posea todos los dispositivos que necesita la tarea. Si no es así, el servicio lanza un error `MissingDevices` que es capturado por el controlador, que genera una respuesta 500. La última comprobación que se realiza es la de la viabilidad de la ejecución del conjunto de tareas que se obtendría al añadir la nueva tarea a las que ya se están ejecutando en el nodo. Tanto para realizar esta comprobación como para llevar a cabo el despliegue como tal, el servicio se apoya en el paquete interno *crane* que ya fue introducido al inicio de esta sección. La función `check_feasibility` de este paquete es comprueba esta viabilidad. Esta función recibe el conjunto de tareas y el número de núcleos de la CPU del nodo objetivo. Con estos datos, se calcula la utilización de CPU para determinar si la ejecución es viable o no [50]. Cabe destacar que, si el nodo tiene más de un núcleo (multiprocesador), esta comprobación de la utilización no es condición suficiente para asegurar la viabilidad del conjunto de tareas, aunque en esta primera versión de la herramienta no se gestiona esta situación.

  Si la función `check_feasibility` determina que el conjunto de tareas es viable, entonces se ejecuta otra función del paquete *crane* para realizar el despliegue: `deploy_task`. Esta función recibe como entrada el fichero `tar.gz` de la tarea, así como sus metadatos y los del nodo. Con estos datos, se establece una conexión con el demonio Docker del nodo mediante SSH, obteniendo un cliente con el que se construye la imagen a partir del fichero `tar.gz` y se lanza el contenedor. Los parámetros de planificación de la tarea (tiempo de ejecución, límite de tiempo y período) se le pasan al contenedor como variables de entorno, las cuáles luego usará la librería para fijar la planificación.

  Cualquier tipo de excepción que se lance durante este despliegue será capturada por el controlador para generar una respuesta 500. Si la operación se realiza correctamente, el servidor responde con una respuesta 200 que contiene en su cuerpo los detalles del nodo actualizado, incluyendo ya en su lista de tareas la que se acaba de añadir. Todo este proceso que se acaba de describir para realizar la operación de despliegue se puede observar de manera esquematizada en el diagrama de secuencias de la figura 4.4.

- `DELETE /nodes/<node_id>/tasks/<task_id>`

  Esta operación permite parar la ejecución de una tarea desplegada en un nodo concreto, elmininándola del mismo. Las peticiones recibidas para esta operación contienen en la

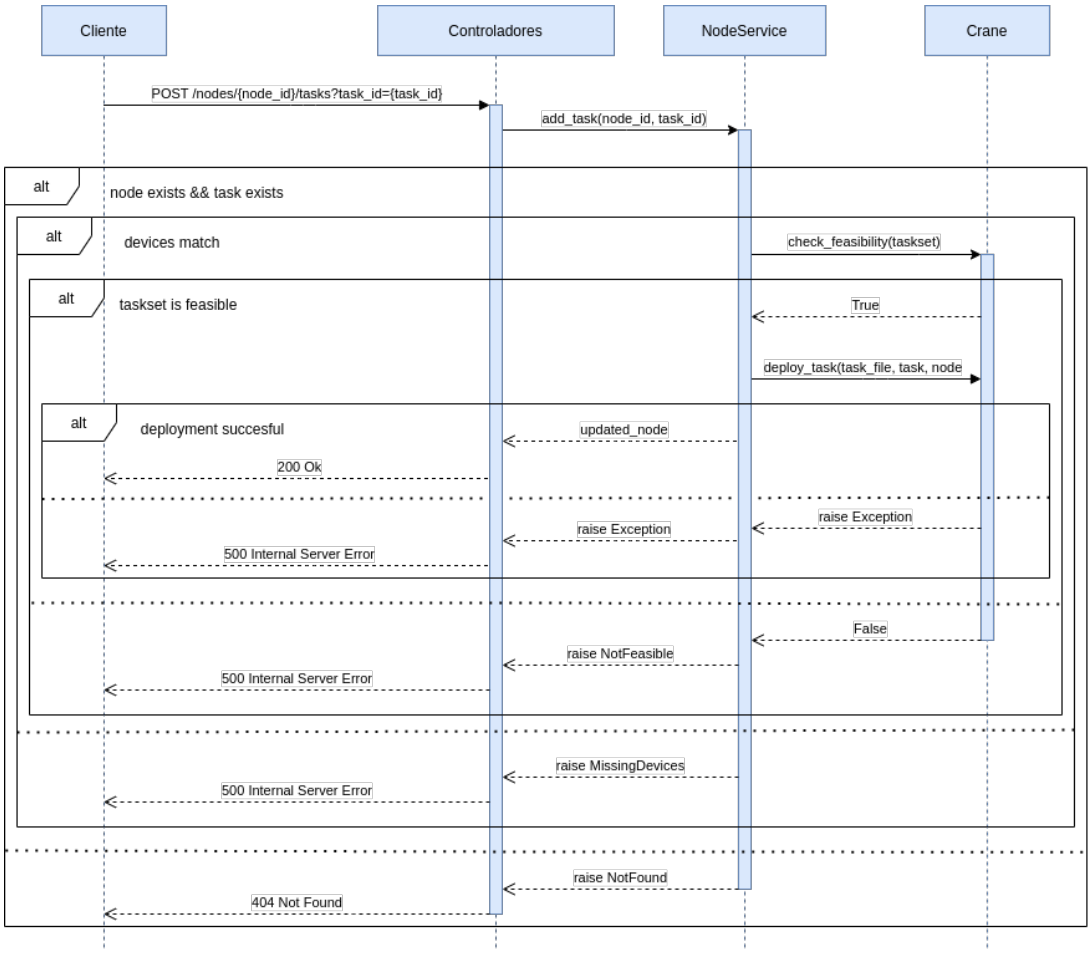


Figura 4.4.: Diagrama de secuencias de la operación de despliegue de una tarea.

ruta los valores de los IDs del nodo y la tarea en cuestión. El controlador entonces llama a la función del servicio encargada de realizar la operación.

Lo primero que hace el servicio es comprobar si están presentes en la base de datos algunos recursos con los IDs proporcionados, lanzando un error `NotFound` en caso de no encontrarlos. Este error es entonces capturado por el controlador, que genera una respuesta 404 para el cliente. Si al buscar los recursos en la base de datos resulta que los IDs no son válidos, se lanza un error `InvalidId`, para el que el controlador crea una respuesta 400.

Si los recursos existen, el servicio entonces ejecuta la función `remove_task` del paquete *crane* para que se lleve a cabo la eliminación. Al igual que ocurría en la de despliegue de tareas, se crea un cliente para el servidor Docker del nodo mediante SSH usando las claves almacenadas. A través de este cliente, se elimina el contenedor en el que se ejecuta la tarea y, también, la imagen asociada. Una vez termina esta operación, se actualiza la base de datos para quitar la tarea de la lista que tiene asociada el nodo. Si durante la eliminación ocurre algún error, el controlador la captura y devuelve una respuesta 500 al cliente. Si la operación se realiza correctamente, la respuesta tiene un código de estado 200 y contiene en su cuerpo los datos actualizados del nodo, ya sin la tarea en su lista de tareas.

Gracias al uso de la librería *hug*, la implementación de los controladores es extremadamente sencilla. Cada controlador es una simple función Python que, mediante decoradores, se enlaza con una operación concreta del servidor a la que da respuesta. En el listado 4.1, se presenta el controlador de la operación de despligue de tareas. Como se puede observar, es con el decorador `@hug.post` con el que indicamos que se debe ejecutar esta función cuando se reciba una petición POST para la ruta `/nodes/<node_id/tasks`[4]. Se definen aquí también las variables de ruta presentes, que en este caso es el ID del nodo.

```
@hug.post('/{node_id}/tasks')
def post_node_tasks(node_id: str, response, task_id: str = None):
    if task_id is None:
        response.status = hug.HTTP_BAD_REQUEST
        return {
            'error': 'No task ID was specified in the request'
        }

    try:
        result = NodeService.add_task(node_id, task_id)
        if result is None:
            response.status = hug.HTTP_INTERNAL_SERVER_ERROR
            return {'error': 'Unable to add task to node.'}
        return Node.Schema().dump(result)
    except InvalidId as e:
        response.status = hug.HTTP_BAD_REQUEST
        return {'error': str(e)}
    except NotFound as e:
```

[4]La ruta `/nodes` es la base para todos los controladores de los nodos, por eso no se muestra en el decorador.

```
        response.status = hug.HTTP_NOT_FOUND
        return {'error': str(e)}
    except (NotFeasible, MissingDevices) as e:
        response.status = hug.HTTP_INTERNAL_SERVER_ERROR
        return {'error': str(e)}
    except Exception:
        response.status = hug.HTTP_INTERNAL_SERVER_ERROR
        return {'error': 'Unable to add task to node.'}
```

Listado de código 4.1: Controlador de la operación de despliegue de tareas.

Las funciones definidas como controladores con *hug* reciben estas variables de ruta como argumentos, teniendo también los posibles parámetros de consulta (`task_id`) y los objetos propios de la comunicación HTTP (p. ej., `request`, `response`, `body`). Observando el listado, se aprecia claramente cómo los controladores se encargan solo de comprobar que el formato de la petición sea el correcto, llamar a la función del servicio apropiada para realizar la operación y construir la respuesta que corresponda en base al resultado de la misma, tal y como se describía anteriormente.

```
@staticmethod
def create(
    new_task: Task,
    file_name: str,
    file_body: BytesIO
    ) -> str:

    result = db.tasks.find_one({'name': new_task.name})
    if result is not None:
        raise AlreadyPresent(
            'A task already exists with the given name.'
        )

    file_id = fs.put(file_body, filename=file_name)
    new_task.file_id = file_id

    new_id = db.tasks.insert_one(Task.Schema(
        exclude=['_id']).dump(new_task)).inserted_id
    return str(new_id)
```

Listado de código 4.2: Función del servicio de tareas para la creación de tareas.

Por otro lado, en el listado 4.2 se muestra la función del servicio de tareas que implementa la lógica encargada de la creación de éstas. El objeto `db`, como se puede suponer, es la conexión con la base de datos que mantiene el servidor. Esta función concreta tiene una particularidad con respecto a las demás: se debe almacenar también el fichero `tar.gz` con el código fuente y la definición de imagen de la tarea. Para ello, usamos GridFS, que es la especificación de MongoDB para el almacenamiento de ficheros de gran tamaño. De esta forma, se pueden almacenar los ficheros en colecciones normales de MongoDB sin necesitar de otro tipo de almacenamiento. Al igual que el objeto `db` era la conexión con la base de datos MongoDB,

`fs` es la conexión con la parte de GridFS. Entonces, para añadir una nueva tarea al sistema, primero se añade el fichero con `fs.put()` y, con el ID recién obtenido para el mismo, se crea el nuevo documento en la colección de tareas.

Otro aspecto a destacar es el uso de `BytesIO` para el acceso al fichero. Cuando el controlador de la operación `POST /tasks` recibe una nueva petición, extrae el contenido del fichero del cuerpo de la misma y lo escribe a un objeto de este tipo. A todos los efectos, un objeto `BytesIO` actúa como un archivo normal, con la diferencia de que sus datos se almacenan directamente en memoria en vez de en el sistema de archivos. Al usar este mecanismo, conseguimos hacer que el proceso de inserción de la tarea sea más rápido, dado que no tenemos que acceder a disco duro ni tenemos que trabajar con ficheros.

Como se ha indicado en la sección 4.2, la API REST usa el formato estándar de codificación de datos JSON para la transmisión de información entre el servidor y los clientes. La mayoría de las peticiones y respuetas contienen en sus cuerpos una representación de los recursos del sistema en este formato. Para que la serialización de los objetos Python a JSON y viceversa se realice de forma más sencilla, se ha usado la librería *marshmallow*. Esta librería nos permite definir esquemas de datos para los recursos del sistema que, además de ayudar con la serialización y deserialización, proporciona un mecanismo de validación automática sobre los datos recibidos de los clientes.

4.3.2. Prueba

Para asegurar el correcto funcionamiento del código implementado, se han escrito pruebas unitarias. Este tipo de pruebas se centran en validar el comportamiento de partes individuales del sistema, aislandolas del resto. En nuestro caso, las pruebas se realizan para las funciones de los controladores y los servicios. La librería *unittest*, que forma parte de la librería estándar de Python, proporciona los mecanismos que se han utilizado para implementar estas pruebas. Con esta librería, se definen clases `TestCase`, que son las unidades individuales de prueba para un componente concreto. Las funciones de estas clases son las que comprueban que las salidas obtenidas para determinadas entradas son las correctas en las funciones individuales del componente dado. Para nuestro servidor, se han definido cuatro clases `TestCase` para los cuatro componentes principales: controladores de los nodos, controladores de las tareas, servicio de nodos y servicio de tareas.

```
def test_post_node_tasks(self):
    response = hug.test.call(
        'POST', controllers, f'{test_nodes[0]._id}/tasks',
        params={
            'task_id': 'Test'
    })
    self.assertEqual(response.status, hug.HTTP_OK)
    self.assertIsNotNone(response.data)

    response = hug.test.call(
        'POST', controllers, f'{ObjectId()}/tasks', params={
            'task_id': 'Test'
```

```
})
self.assertEqual(response.status, hug.HTTP_NOT_FOUND)
self.assertIsNotNone(response.data)
self.assertIsInstance(response.data['error'], str)

response = hug.test.call(
    'POST', controllers, f'{test_nodes[0]._id}/tasks')
self.assertEqual(response.status, hug.HTTP_BAD_REQUEST)
self.assertIsNotNone(response.data)
self.assertIsInstance(response.data['error'], str)

response = hug.test.call(
    'POST', controllers, f'error/tasks', params={
        'task_id': 'Test'
})
self.assertEqual(response.status, hug.HTTP_BAD_REQUEST)
self.assertIsNotNone(response.data)
self.assertIsInstance(response.data['error'], str)

response = hug.test.call(
    'POST', controllers, f'{test_nodes[0]._id}/tasks',
    params={
        'task_id': 'MissingDevices'
})
self.assertEqual(
    response.status, hug.HTTP_INTERNAL_SERVER_ERROR)
self.assertIsNotNone(response.data)
self.assertIsInstance(response.data['error'], str)

response = hug.test.call(
    'POST', controllers, f'{test_nodes[0]._id}/tasks',
    params={
        'task_id': 'NotFeasible'
})
self.assertEqual(
    response.status, hug.HTTP_INTERNAL_SERVER_ERROR)
self.assertIsNotNone(response.data)
self.assertIsInstance(response.data['error'], str)
```

Listado de código 4.3: Prueba unitaria del controlador para el despliegue de tareas.

Como se acaba de explicar, una prueba unitaria valida una única parte del código de forma aislada. En nuestro servidor, tanto los servicios como los controladores deben interactuar con otros componentes para cumplir su función. Para poder probar solo una función concreta, necesitamos hacer maquetas o *mocks* de los demás componentes con los que interactúa. Estas maquetas son implementaciones de prueba, en muchos casos vacías o con un comportamiento concreto para cada caso de prueba. Por ejemplo, para poder probar los controladores, necesitamos una maqueta del servicio con el que trabajan. La maqueta ofrece una implementación «de juguete» con un comportamiento igual al que tendría el servicio real. La librería *unittest* proporciona el decorador `@mock.patch` para las clases `TestCase`, con el cuál se puede propor-

cionar una maqueta para que sea usada por el código que se prueba en lugar del elemento real.

En el listado 4.3 se puede ver la función de prueba para el controlador de despliegue de tareas del `TestCase` de los controladores de los nodos. En 4.1 se puede ver como este controlador llama a la función `add_task` del servicio de nodos. La maqueta del servicio que se usa para esta prueba tiene una implementación simple de esta función que, dependiendo de la entrada dada, permite obtener todas los resultados que se podrían obtener con la implementación real. De esta forma, podemos comprobar que el controlador se comporte correctamente para cada una de estas salidas.

Volviendo a la prueba de 4.3, consiste sencillamente de una serie de peticiones con distintos valores que se envían al controlador gracias a la función `hug.test.call()` que la propia librería *hug* proporciona para escribir pruebas. Para cada petición, se comprueba que la respuesta tiene el código de estado HTTP correcto, además de los valores deseados en el cuerpo.

```
def test_create(self):
    new_task = Task.Schema().load({
        'name': 'Test',
        'runtime': 1000,
        'deadline': 1000,
        'period': 1000
    })

    try:
        result = TaskService.create(
            new_task, 'test_file.tar.gz', BytesIO())
        self.assertEqual(
            mockdb.tasks.count_documents({}), len(test_tasks)+1)
        self.assertIsInstance(result, str)
    except:
        self.fail()

    with self.assertRaises(AlreadyPresent):
        TaskService.create(
            new_task, 'test_file.tar.gz', BytesIO())
    self.assertEqual(
        mockdb.tasks.count_documents({}), len(test_tasks)+1)
```

Listado de código 4.4: Prueba unitaria de la función de creación de tareas.

Los servicios, por otro lado, interactúan tanto con la base de datos MongoDB como con las funciones de la librería interna *crane*. Estas funciones se pueden maquetar al igual que hacíamos con las de los propios servicios para probar los controladores dado que conocemos su implementación y su alcance es pequeño. La conexión con la base de datos plantea una situación diferente. Podríamos levantar una instancia de MongoDB cada vez que quisiéramos ejecutar las pruebas, aunque esto rompe ligeramente con el enfoque aislado de las pruebas unitarias. Además, esto tiene un impacto considerable en la complejidad de la ejecución de estas pruebas, que deberían ser simples y directas. Por ello, hacemos uso de la librería *mongomock*,

que nos proporciona una maqueta de los clientes de MongoDB usados por los servicios. Estas maquetas funcionan exactamente igual que los clientes normales, almacenando los datos de la hipotética base de datos en memoria y permitiendo comprobar que las operaciones se realizan correctamente. Antes de ejecutar las pruebas de los servicios, se añaden datos de prueba a estas maquetas, de forma que se simule una base de datos real en producción.

Por lo demás, las pruebas como tal son muy similares a las de los controladores, tal y como se puede ver en el listado 4.4. En él, se muestra la validación del funcionamiento de la lógica de creación de tareas que fue expuesta en 4.2, que consiste en añadir una tarea simple y comprobar que en la base de datos maquetada (`mockdb`) se ha añadido el nuevo documento. Se repite la inserción para asegurar que no se pueden crear elementos duplicados.

4.3.3. Integración continua

Para facilitar el despliegue de este servidor en múltiples plataformas, se ha definido una imagen Docker sencilla. Esta imagen se puede encontrar en el repositorio de DockerHub[5]. La construcción de la imagen se realiza de forma automática cuando se publica una nueva versión del servidor en el repositorio de GitHub del mismo gracias a GitHub Actions. Esta funcionalidad de GitHub permite definir secuencias de pasos que se ejecutan cuando ocurren ciertos eventos en el repositorio.

Además de construir la imagen y publicarla en DockerHub para cada nueva *release*, se ha definido otra acción que ejecuta las pruebas unitarias del proyecto de forma automática para cada nuevo cambio que se realiza sobre la base de código del repositorio. Al ejecutar las pruebas, se obtiene además un informe de la cobertura del código que se envía a Codecov. La cobertura mide el porcentaje de líneas de código del proyecto que son ejecutadas por alguna prueba. Codecov permite visualizar estos informes de manera sencilla con paneles de control como el mostrado en la figura 1.3. De esta forma, podemos monitorizar la calidad del código y comprobar si los nuevos cambios introducidos siguen pasando las pruebas de forma correcta.

4.4. Cliente de terminal

El diseño detallado en la sección 4.2 especificaba que todas las funcionalidades del sistema se recogen en un servidor maestro que expone las distintas operaciones a través de una API REST. Este modelo cliente-servidor permite la implementación de múltiples clientes para plataformas diferentes de forma sencilla. En nuestro caso, se ha decidido acompañar esta primera versión del sistema con un cliente de línea de comandos que permita llevar a cabo las distintas operaciones existentes.

Aunque un cliente con interfaz gráfica es más accesible y fácil de usar, se ha decidido hacerlo de terminal debido a que es más versátil: fácilmente se puede incorporar en *scripts* de `bash` y tareas automáticas de `cron`, además de ser usado con herramientas de integración continua modernas.

[5]Enlace a la imagen: `https://hub.docker.com/r/varrrro/shipyard-server`

4.4.1. Diseño e implementación

De forma idéntica a como sucedía con el servidor, la lógica del cliente se ha dividido por dominio y responsabilidades. En este caso, la división por dominio se realiza en tres partes: operaciones de nodos, de tareas y de orquestación. Para cada uno de estos dominios, tenemos dos componentes separados: los comandos y los servicios. Los primeros, como su propio nombre indica, definen los comandos que puede ejecutar el usuario desde la terminal, indicando todos los argumentes necesarios y las opciones disponibles. Los comandos validan la información introducida por el usuario, llamando después a los servicios para llevar a cabo las operaciones. Estos servicios son, al igual que ocurría con el servidor, clases con métodos estáticos que implementan la lógica de negocio de la aplicación. En el cliente, los servicios se encargan de realizar las peticiones HTTP al servidor y procesar las respuestas. Según la respuesta recibida, las funciones de los servicios pueden devolver los datos esperados o lanzar errores, en base a los cuáles el comando mostrará una respuesta u otra en la terminal para el usuario.

A continuación, se presentan y detallan todos los comandos que se han definido para esta primera versión del cliente. Todas las operaciones comienzan con el comando base `shipyard`, que es el nombre que se le ha dado al sistema.

- `shipyard node ls`

  Con este comando, se obtiene del servidor una lista con todos los nodos presentes en el sistema. El cliente imprime en la terminal una tabla con el ID, nombre, dirección IP y número de tareas en ejecución de cada nodo.

  Si se usa la opción `--active`, solo se muestran aquellos nodos en los que se está ejecutando al menos una tarea.

- `shipyard node inspect <key>`

  En este caso, se buscan los detalles de un nodo concreto, identificado por la clave dada. Esta clave puede ser tanto el ID del nodo como su nombre. Es la función del comando la encargada de comprobar si se trata de un ID válido para llamar a la función `get_by_id` del servicio, llamando a `get_by_name` si no lo es.

  El comando imprime en la terminal es su lista de tareas, mostrando el ID, nombre, tiempo de ejecución (*runtime*), límite de tiempo (*deadline*), período (*period*) de cada una de ellas. Si el servidor responde con un error (p. ej., si no se encuentra ningún nodo con la clave dada), se indica al usuario.

- `shipyard node add <name> <address> <cpu-cores>`

  Con este comando, el usuario puede añadir un nuevo nodo al sistema. En la ejecución del propio comando, se indican el nombre, la dirección IP y el número de núcleos del CPU del nuevo nodo. Inmediatamente después de introducir el comando en la terminal, la aplicación pide al usuario que introduzca el nombre de usuario y la contraseña para realizar la conexión SSH inicial con el dispositivo y copiar las claves.

  Si el usuario desea especificar más valores para otros atributos del nodo, puede usar

las opciones disponibles para el comando: `--cpu`, `--cpu-arch`, `--cpu-freq`, `--ram` y `--device`.

En pantalla se imprime el ID del nuevo nodo si la operación se ha realizado correctamente, de forma que el usuario pueda realizar más operaciones con él. Si la respuesta del servidor no es correcta, se imprime el mensaje de error adecuado por la terminal para informar al usuario.

- `shipyard node update <key> <values>`

  Este comando se puede usar para cambiar los atributos de un nodo ya presente en el sistema. Al igual que ocurría con el comando `node inspect`, el nodo se puede identificar tanto por su ID como por su nombre. El argumento *values* del comando debe ser una cadena de texto en formato JSON con los pares clave-valor de los atributos a actualizar. Si el formato de la cadena proporcionada no es correcto, se imprime un mensaje de error para el usuario y la operación se cancela. En caso de ser correcto, se llama a la función del servicio encargada de realizar la petición de modificación.

  Se debe tener en cuenta que, si el nombre del nodo o su lista de dispositivos son modificados, realizar esta operación conlleva la suspensión de la ejecución de todas las tareas que hubiera en dicho nodo. Tanto si la operación se realiza correctamente como si no, se muestra un mensaje en la terminal para informar al usuario del resultado.

- `shipyard node rm <key>`

  Un usuario puede eliminar un nodo usando este comando. Una vez más, se puede identificar el nodo a eliminar tanto por su ID como por su nombre. Al ejecutar este comando, el usuario debe confirmar que desea realizar la operación para evitar accidentes. Al terminar la operación, se muestra un mensaje por la terminar para informar al usuario del resultado.

- `shipyard task ls`

  Al igual que el comando `node ls`, este comando sirve para obtener una lista de las tareas almacenadas en el sistema. Se imprime en la pantalla una tabla que muestra el ID, nombre, tiempo de ejecución, límite de tiempo y período de cada tarea.

- `shipyard task add <name> <runtime> <deadline> <period> <file>`

  Este comando es el usado para añadir una nueva tarea al sistema. En los argumentos, se indican los valores necesarios para la tarea: nombre, tiempo de ejecución, límite de tiempo, período y el fichero `tar.gz` que contiene los desplegables de la tarea. Si el usuario desea especificar también una serie de dispositivos que necesita la tarea, puede hacerlo con la opción `--device` e indicar tantos como quiera. Si la operación se realiza correctamente, el ID de la nueva tarea se imprime por la terminal. Por el contrario, si ocurre algún error que impida la realización de la operación, se informa al usuario.

- `shipyard task update <key> <values>`

  Un usuario puede actualizar una tarea usando este comando. Al igual que ocurría con el

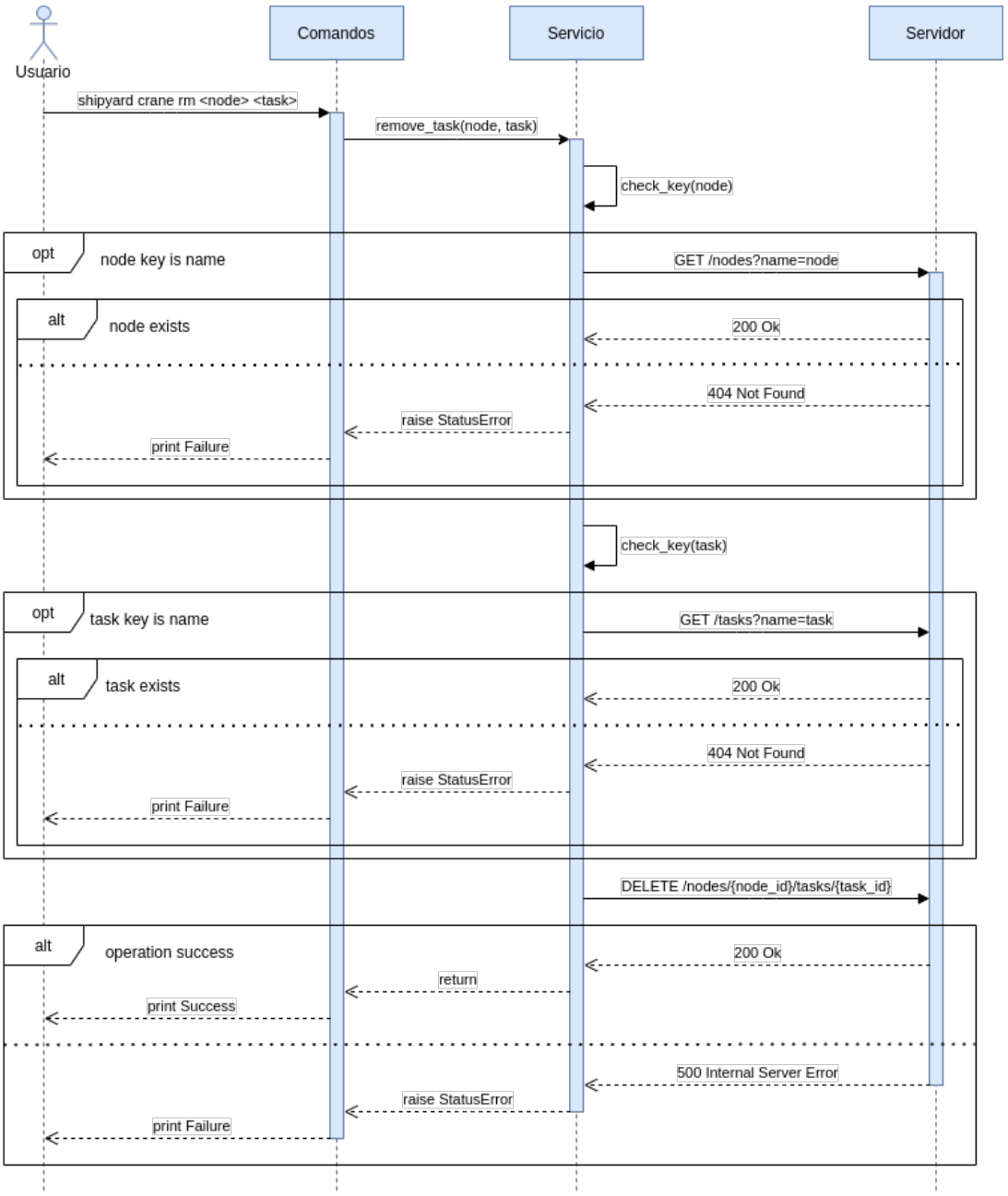


Figura 4.5.: Diagrama de secuencias del comando de de eliminación de una tarea de un nodo.

comando `node update`, se pasan como argumentos la clave que identifica la tarea que se debe actualizar y los nuevos valores de los atributos en formato JSON. La clave puede ser tanto el ID de la tarea como su nombre. Si el usuario desea actualizar también el fichero `tar.gz` asociado a la tarea, debe usar la opción `--file` junto con la ruta del nuevo fichero.

Tanto si la operación se realiza correctamente como si no, se imprime por pantalla un mensaje para informar al usuario del resultado. Cabe destacar que la operación del servidor a la que se llama para actualizar una tarea provoca la eliminación de dicha tarea de todos los nodos en los que se estuviera ejecutando.

- `shipyard task rm <key>`

  Con este comando, el usuario puede eliminar una tarea del sistema. El argumento *key* es la clave con la que se identifica la tarea a eliminar, ya sea su ID o su nombre. Al eliminar una tarea, la operación del servidor a la que se llama se encarga de eliminarla de todos los nodos en los que se estuviera ejecutando. Tanto si la operación se realiza correctamente como si no, se imprime por pantalla un mensaje para informar al usuario del resultado.

- `shipyard crane deploy <node-key> <task-key>`

  Esta es, quizás, la operación más importante que se puede realizar desde el cliente. Permite a un usuario desplegar una tarea del sistema en un nodo concreto. El nodo y la tarea en cuestión se identifican por las claves pasadas como argumentos del comando. Estas claves pueden ser tanto sus IDs como sus nombres. La función del comando entonces llama a la función del servicio *crane* encargada de enviar la petición HTTP al servidor para que se realice la operación. Se informa al usuario del resultado de la operación mediante un mensaje que se imprime en la terminal.

- `shipyard crane rm <node-key> <task-key>`

  Al igual que el usuario puede desplegar tareas en los nodos, también puede retirarlas de estos usando este comando. De nuevo, el nodo y la tarea concretos son identificados por las claves pasadas como argumentos en la llamada al comando y pueden ser tanto sus IDs como sus nombres. Después, la función del servicio *crane* se encarga de realizar la petición HTTP apropiada y procesar la respuesta, informando el comando al usuario del resultado mediante un mensaje impreso en la terminal. En la figura 4.5 se muestra en un diagrama de secuencias la lógica de ejecución de este comando.

- `shipyard config <key> <value>`

  Este comando permite cambiar la configuración de la aplicación. Se debe indicar en los argumentos el parámetro a cambiar (*key*) y el nuevo valor (*value*). Se usa principalmente para especificar la dirección del servidor maestro.

Definir los comandos es relativamente sencillo gracias a la librería *click*. Sencillamente, definimos una función Python para cada comando y usamos decoradores para indicar que es un comando y definir los argumentos y opciones, de forma similar a lo que hacíamos con *hug* para

definir los controladores de la API REST del servidor. En el listado de código 4.5 se muestra la función implementada para el comando `shipyard task add`. Vamos a explicar qué significa cada decorador usado:

- `@task.command`: Normalmente, se usa el decorador `@click.command` para indicar que una función es un comando de la aplicación. Aquí, se usa `task` debido a que es un subcomando del comando base con el mismo nombre.
- `@click.pass_obj`: Usado para pasar el contexto de la aplicación, que contiene el servicio apropiado (en este caso, el de tareas), a la función.
- `@click.option`: Son las opciones no requeridas que se pueden indicar en la ejecución del comando. Pueden ser de cualquier tipo y, como ocurre en el ejemplo mostrado, de uso múltiple.
- `@click.argument`: Los argumentos obligatorios del comando. Pueden ser de cualquier tipo.

Como se puede apreciar en el listado; el contexto, las opciones y los argumentos son también argumentos de la función. Entrando ya en la implementación de la operación como tal, se puede ver como se crea el objeto `Task` con los datos dados usando un esquema definido con la librería *marshmallow*. Igual que ocurría en el servidor, usamos esta librería para facilitar el trabajo con los recursos, haciendo muy sencillo el paso de objeto Python a diccionario y viceversa.

```
@task.command()
@click.pass_obj
@click.option(
    '--device', type=str, multiple=True,
    help='A device needed for the task.'
)
@click.argument('name', type=str)
@click.argument('runtime', type=int)
@click.argument('deadline', type=int)
@click.argument('period', type=int)
@click.argument('file', type=click.File('rb'))
def add(service, device, name, runtime, deadline, period, file):
    try:
        task = Task.Schema().load({
            'name': name,
            'runtime': runtime,
            'deadline': deadline,
            'period': period,
        })
        task.devices = list(d for d in device)
        new_id = service.create(task, file)
        click.echo(f'Added new task with ID {new_id}')
    except Exception as e:
        click.echo('Unable to add new task:\n' + str(e))
```

Listado de código 4.5: Comando para la creación de una nueva tarea.

En los servicios, se hace uso de la librería *requests* para realizar las peticiones HTTP al servidor. La función del servicio de nodos encargada de su modificación se puede ver en 4.6. Aquí, se aprecia cómo se construyen estas peticiones con los disintos verbos HTTP (`requests.get` y `requests.put`) y se gestionan las respuestas recibidas, comprobando su código de estado para devolver un resultado o lanzar el error correspondiente.

```
def update(self, node_key: str, new_values: dict) -> Node:
    if not bson.ObjectId.is_valid(node_key):
        response = requests.get(
            self.base_url + '/nodes?name=' + node_key)
        if response.status_code != HTTPStatus.OK:
            raise StatusError(response.json()['error'])
        node_key = response.json()['_id']

    response = requests.put(
        self.base_url + '/nodes/' + node_key,
        json=new_values
    )
    if response.status_code == HTTPStatus.OK:
        return Node.Schema().load(response.json())
    else:
        raise StatusError(response.json()['error'])
```

Listado de código 4.6: Función del servicio de nodos para su modificación.

4.4.2. Prueba

Para el cliente también se han implementado algunas pruebas unitarias que validan su comportamiento, aunque solo de los comandos. Al igual que *hug*, *click* proporciona mecanismos para la prueba de los comandos implementados. En el listado 4.7 se muestra el método del *TestCase* de los comandos de las tareas encargado de preparar la ejecución de las pruebas. Como se puede apreciar, se usa un objeto `CliRunner` que simula la ejecución por consola. También se crea una configuración de prueba. Además de esto, también es necesario maquetar el servicio correspondiente, igual que ocurría con las pruebas de los controladores del servidor.

```
@classmethod
def setUpClass(self):
    self.runner = CliRunner()
    self.config = Config(server_url='test', server_port='test')
```

Listado de código 4.7: Función preparatoria para las pruebas de los comandos de las tareas.

Las funciones de prueba como tal usan este `CliRunner` para simular la ejecución de los comandos. En el listado 4.8 se puede ver como se utiliza para probar el comando de creación de tareas. Con la función `invoke`, se ejecuta el comando con distintos argumentos y opciones y se comprueba que la salida es la correcta.

```
def test_add(self):
    with self.runner.isolated_filesystem():
        with open('test.tar.gz', 'wb') as f:
            f.write(str.encode('Test'))

        result = self.runner.invoke(
            task,
            ['add', 'Test', '1000', '1000', '1000',
                'test.tar.gz'],
            obj=self.config
        )
        self.assertIn('Added new task', result.output)

        result = self.runner.invoke(
            task,
            ['add', '--device', '/dev/null', 'Test',
                '1000', '1000', '1000', 'test.tar.gz'],
            obj=self.config
        )
        self.assertIn('Added new task', result.output)

        result = self.runner.invoke(
            task,
            ['add', 'Test1', '1000', '1000', '1000',
                'test.tar.gz'],
            obj=self.config
        )
        self.assertIn('Unable to add new task', result.output)
```

Listado de código 4.8: Prueba del comando de creación de una nueva tarea.

4.4.3. Integración continua

Las pruebas que se acaban de describir también se ejecutan de forma automática cuando se realiza algún cambio en la base de código del cliente mantenida en su repositorio de GitHub. Un flujo de trabajo casi idéntico al usado en el servidor se encarga de lanzar estas pruebas y enviar el informe de cobertura generado a Codecov.

No obstante, el cliente no se distribuye como una imagen Docker, como sí ocurría con el servidor. Se ha optado mejor por publicarlo como un paquete de PyPI[6], que es el repositorio

[6]Enlace al paquete en PyPI: `https://pypi.org/project/shipyard-cli/`

de software principal de Python, donde se encuentra la gran mayoría de paquetes y con el que trabaja el gestor de dependencias `pip` por defecto. Para poder publicar el paquete en esta plataforma, se añade un fichero `setup.py` a la raíz del proyecto con los atributos del mismo (p. ej., autor, descripción, dependencias, ...). Luego, se define una acción en el repositorio de GitHub que se encarga de construir el artefacto y publicarlo en PyPI cada vez que se publique una nueva versión. Gracias a esto, el cliente se puede instalar tan solo ejecutando `pip install shipyard-cli`.

4.5. Imagen base

Normalmente, al implementar un proceso de tiempo real en Linux, el desarrollador dedica una parte del código a fijar la política y los parámetros de planificación deseados. En nuestro sistema, si dejáramos esta responsabilidad en los desarrolladores de las tareas, se podrían producir inconsistencias entre los parámetros indicados al añadir la tarea y los usados finalmente en la planificación de la misma, además de no poder el sistema controlar y ajustar estos valores.

Para evitar esto, se proporciona una librería C en una imagen Docker que debe ser usada como imagen base por todas las tareas que se quieran desplegar con el sistema.

4.5.1. Diseño e implementación

Como ya se indicó en la sección 2.4, la política de planificación `SCHED_DEADLINE` de Linux para procesos de tiempo real es la usada en nuestro sistema debido a ser dinámica y adaptarse a nuestras necesidades. Aunque esta política forma parte del kernel de Linux desde la versión 3.14, las funciones y estructuras necesarias para implementar procesos que usen esta planificación no son todavía ofrecidas por las librerías estándar. Un usuario que desee usar `SCHED_DEADLINE` para ejecutar procesos debe proporcionar la implementación de las funciones `sched_setattr` y `sched_getattr`, además de la estructura `sched_attr`. Aunque esta implementación es simple, tener que incorporarla para cada proceso es una inconveniencia. Por ello, se ha decidido implementar estos elementos en la librería que se ofrece a los usuarios para apoyar el desarrollo de tareas que se desplieguen dentro del sistema *shipyard*.

Como ya se ha indicado al principio de esta sección, el objetivo principal de la librería no es realmente proporcionar estas funciones, las cuáles ni siquiera son accesibles desde el exterior de la misma, sino establecer el mecanismo mediante el cuál las tareas del sistema definen sus atributos de planificación y se ejecutan. La librería expone una única función, `set_sched`, cuya implementación se puede ver en el listado 4.9 y que debe ser ejecutada al inicio de cada tarea que se despliega.

```
int set_sched()
{
    struct sched_attr attr;
    int runtime;
    int deadline;
```

```
    int period;
    int ret;

    runtime = atoi(getenv("TASK_RUNTIME"));
    deadline = atoi(getenv("TASK_DEADLINE"));
    period = atoi(getenv("TASK_PERIOD"));

    attr.size = sizeof(attr);
    attr.sched_flags = 0;
    attr.sched_nice = 0;
    attr.sched_priority = 0;
    attr.sched_policy = SCHED_DEADLINE;
    attr.sched_runtime = runtime;
    attr.sched_deadline = deadline;
    attr.sched_period = period;

    ret = sched_setattr(gettid(), &attr, 0);
    if (ret < 0)
    {
        return -1;
    }

    return 0;
}
```

Listado de código 4.9: Función encargada de fijar los parámetros de planificación.

Observando la función, se aprecia como ésta sencillamente obtiene los valores del tiempo de ejecución, el límite de tiempo y el período para la tarea de variables de entorno y, luego, los incorpora a una estructura `sched_attr` con la que establece la planificación llamando a la función `sched_setattr`.

Aunque es suficiente con aportar la librería para que los usuarios del sistema implementen sus tareas, hemos decidido proporcionar esta librería dentro de una imagen Docker que las imágenes de estas tareas puedan usar como base. El proceso planteado para la ejecución de las tareas es el siguiente:

1. Al ejecutar la operación de despliegue, el servidor lanza en el nodo el contenedor de la tarea, pasándole los parámetros de planificación de la misma como variables de entorno.

2. El proceso del usuario importa la librería y ejecuta la función `set_sched`.

3. Se fija la planificación extrayendo los parámetros de las variables de entorno.

De esta forma, aseguramos que el sistema tiene el control sobre la planificación de las tareas y evitamos problemas de inconsistencia.

En el listado de código 4.10 se muestra la definición de la imagen Docker de la librería. Básicamente, en la construcción de la imagen se copian los ficheros de la librería y se compila, obte-

niendo un objeto estático que luego será enlazado por la tarea durante su compilación. Además, se definen tres variables de entorno dentro del contenedor: `SRC_DIR`, `LIB_DIR` y `LIB_NAME`. Estas variables contienen las rutas de los directorios con el código fuente y los artefactos compilados, además del nombre de la librería, respectivamente. Son utilizadas en las imágenes de las tareas de los usuarios para poder enlazar la librería en la compilación.

```
FROM gcc:10

# Change dir and copy files
WORKDIR /shipyard
COPY src/ src/
COPY Makefile .

# Build static library
RUN make

# Set up env vars and change dir for child images
ENV SRC_DIR /shipyard/src
ENV LIB_DIR /shipyard/build
ENV LIB_NAME shipyard
WORKDIR /task
```

Listado de código 4.10: Definición de la imagen Docker para la librería.

Un ejemplo de `Dockerfile` para una tarea simple lo podemos ver en el listado 4.11. Las variables de entorno `SRC_DIR`, `LIB_DIR` y `LIB_NAME` se usan tal y como acabamos de comentar. La imagen `gcc:10` en la que se basa la imagen de la librería es multiarquitectura, de forma que las imágenes basadas en ella se pueden construir y ejecutar en multitud de plataformas diferentes.

```
FROM varrrro/libshipyard

COPY task.c .

RUN gcc -o task task.c -L$LIB_DIR -I$SRC_DIR -l$LIB_NAME

CMD [ "./task" ]
```

Listado de código 4.11: Ejemplo de definición de imagen para una tarea del sistema.

4.5.2. Integración continua

De manera muy similar a lo que sucedía con el servidor, para la librería también se ha definido una acción de GitHub que se encarga de construir de manera automática la imagen Docker y publicarla en DockerHub[7] para cada nueva *release* de la librería que se realice en el repositorio.

[7]Enlace a la imagen en DockerHub: `https://hub.docker.com/r/varrrro/libshipyard`

La única diferencia radica en que la imagen de la librería es multiplataforma, como ya se ha comentado en la subsección anterior, lo que hace que sea ligeramente diferente el proceso de construcción.

Para poder construir imágenes multiplataforma, se debe hacer uso de la nueva utilidad Buildx de Docker. Adortunadamente, Docker ofrece tareas de GitHub Actions ya preparadas para usar este nuevo método de construcción, con lo que solo tenemos que especificar las plataformas deseadas en la tarea. Para esta nueva versión, se soportan solo amd64, arm64 y armv7, ya que son a las que se ha tenido acceso para su validación, aunque en teoría la imagen debería funcionar a la perfección en todas las plataformas en las que funciona la imagen `gcc:10` en la que se basa.

4.6. Análisis del rendimiento

Para completar la presentación de esta primera versión de la herramienta, se ha querido realizar una breve caracterización de su rendimiento. En concreto, lo que se ha medido es la latencia en el despliegue de las tareas, es decir, el tiempo que pasa desde que se inicia la operación de despliegue hasta que la tarea comienza a ejecutarse en el nodo objetivo.

blablabla faltan los resultados

A. Estimación de costes del proyecto

Al tratarse, mayoritariamente, de un proyecto de Ingeniería del Software, se ha realizado también una estimación de los costes asociados a la realización del mismo. Para ello, se ha aplicado un modelo basado en tiempo y materiales. En 2019, el coste salarial medio de un ingeniero informático recién salido de la universidad en España era de entre 24.000 y 28.500 euros brutos al año. En esta estimación, se ha asumido un salario mensual de 2.000€ brutos al mes, que es aproximadamente un salario anual de 24.000€. Las tareas de coordinación y dirección del jefe del proyecto se estiman en un 10 % del trabajo de ingeniería, con un coste medio mensual de unos 5.000€ al mes para un ingeniero sénior. Además, se ha supuesto también una jornada laboral que llega al máximo en España de 40 horas semanales. También se tiene en cuenta el coste del hardware usado para pruebas. La estimación final del coste de desarrollo de este proyecto se muestra en la tabla A.1.

Raspberry Pi 4B 4GB	60,00€
Cable Ethernet	7,00€
Cable de alimentación USB-C	7,00€
Mano de obra ingeniero	8.000,00€
Mano de obra ingeniero jefe	20.000,00€
	28.074,00€

Tabla A.1.: Desglose de costes del proyecto.

A todo esto habría que sumarle el coste de la estación de trabajo usada para el desarrollo del proyecto, la cuál consiste de un ordenador de sobremesa o portátil y los periféricos necesarios, junto con los gastos asociados al consumo eléctrico y el acceso a internet. Estos gastos se han omitido de la estimación realizada debido a que son muy variables y tampoco tienen un impacto muy representativo en los costes del proyecto.

B. Instalación del kernel de Linux con PREEMPT_RT en Raspberry Pi

Normalmente, para obtener un kernel de Linux parcheado con PREEMPT_RT es necesario descargarse por separado tanto la versión del kernel que queramos usar como la versión del parche adecuada para dicho kernel. Luego, se aplica el parche, se realizan algunos ajustes en la configuración para el dispositivo específico en el que se va a usar y se compila el nuevo kernel. Se trata de un proceso relativamente sencillo, pero que requiere de conocimientos avanzados sobre el kernel de Linux. Por suerte, la comunidad de usuarios de Raspberry Pi mantiene de forma pública versiones del kernel parcheadas con PREEMPT_RT para estos dispositivos, lo que facilita enormemente el proceso de instalación. En este apéndice, se describe el proceso a seguir para instalar en una Raspberry Pi 4B una de estas versiones del kernel de Linux parcheado para tiempo real.

Cabe destacar que, aunque la Raspberry Pi es de arquitectura ARM, el kernel se puede preparar en un ordenador independiente que sea x86, que es lo que se ha hecho para este trabajo y lo que se explica en este apéndice. En este caso, lo que se realiza es una compilación cruzada.

Antes de comenzar, es necesario crear los directorios sobre los que se va a trabajar y clonar los repositorios de GitHub donde se guardan tanto las versiones del kernel como las herramientas necesarias para su compilación. En el caso del repositorio que contiene los kernels, clonamos la rama de la versión concreta que queremos usar. Las ramas que contienen versiones parcheadas para tiempo real se identifican con el sufijo `-rt`. Tal y como se puede ver en el listado B.1, hemos usado la versión 4.19 del kernel, dado que es la más reciente que se encuentra disponible con el parche en el momento en el que se ha realizado el trabajo.

```
mkdir -p ~/rpi-rt/rt-kernel && cd ~/rpi-rt
git clone https://github.com/raspberrypi/linux.git
    -b rpi-4.19.y-rt
git clone https://github.com/raspberrypi/tools.git
```

Listado de código B.1: Preparación de las carpetas y descarga de los repositorios.

Antes de compilar, en el listado B.2 se muestra como se guarda en variables de entorno algunos valores que serán necesarios para realizar esta compilación.

```
export ARCH=arm
export CROSS_COMPILE=~/rpi-rt/tools/arm-bcm2708/
    gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin/
    arm-linux-gnueabihf-
export INSTALL_MOD_PATH=~/rpi-rt/rt-kernel
```

```
export INSTALL_DTBS_PATH=~/rpi-rt/rt-kernel
export KERNEL=kernel71
```

Listado de código B.2: Definición de variables de entorno.

Una vez hecho esto, procedemos con la compilación. Para ello, se deben ejecutar los objetivos del `Makefile` ubicado en `\mytilde/rpi-rt/linux` y que se muestran en el listado B.3. En este listado, el valor `X` que se le pasa a la opción `-j` debe ser igual al número de núcleos de CPU que posea el ordenador en el que se está realizando la compilación, con el objetivo de que se haga más rápido.

```
make bcm2711_defconfig
make -jX zImage
make -jX modules
make -jX dtbs
make -jX modules_install
make -jX dtbs_install
```

Listado de código B.3: Compilación del kernel parcheado.

Cuando termina la compilación, debemos añadir a la nueva imagen la información relativa al proceso de inicio del dispositivo o *boot*, ejecutando los comandos que se muestran en el listado B.4.

```
mkdir $INSTALL_MOD_PATH/boot
./scripts/mkknlimg ./arch/arm/boot/zImage
        $INSTALL_MOD_PATH/boot/$KERNEL.img
cd $INSTALL_MOD_PATH/boot
mv $KERNEL.img rt_kernel.img
```

Listado de código B.4: Añadir información de *boot*.

Todos los archivos de la carpeta `\mytilde/rpi-rt/rt-kernel` se comprimen entonces en un `tar.gz` que se copia a la Raspberry Pi, a la cuál debemos acceder ahora para ejecutar los comandos del listado B.5, que son los que consiguen instalar el nuevo kernel.

```
tar -xzf rt-kernel.tar.gz
cd boot && sudo cp -rd * /boot/
cd ../lib && sudo cp -rd * /lib/
cd ../overlays && suco cp -d * /boot/overlays
cd .. && sudo cp -d bcm* /boot/
```

Listado de código B.5: Instalación del kernel.

Para terminar, añadimos `kernel=rt_kernel.img` al fichero `/boot/config` y reiniciamos la Raspberry Pi. Tras el reinicio, el comando `uname -r` nos tiene que indicar la nueva versión del

kernel, que, en nuestro caso, es `4.19.71-rt24-v7l+`. Ahora, el kernel puede realizar planificación apropiativa, además de recibir cambios en otros mecanismos que reducen su latencia, con lo que la Raspberry Pi puede ejecutar tareas de tiempo real.

C. Ejecución de las pruebas de `rt-tests` dentro de contenedores

Como se ha visto en el capítulo 3, para estudiar el impacto que tiene el uso de contenedores sobre la latencia en la activación de procesos de tiempo real se ha hecho uso de la utilidad `cyclicdeadline` de la suite de pruebas de tiempo real para Linux `rt-tests`. Para obtener los resultados deseados, era necesario ejecutar estas pruebas tanto de forma «nativa» como dentro de contenedores. En este apéndice se expone la imagen Docker que se ha preparado para ejecutar no solo `cyclicdeadline`, sino cualquier prueba de `rt-tests` dentro de un contenedor[1]. En el listado C.1 se puede ver el `Dockerfile` con esta definición.

```
FROM debian:stable-slim

RUN apt update && apt install -y \
        git \
        build-essential \
        bison \
        flex \
        libnuma-dev \
        python3 \
        python3-distutils

RUN git clone \
    git://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git
WORKDIR ./rt-tests

RUN git checkout tags/v1.8 -b v1.8
RUN make all && make install

VOLUME [ "/sys/kernel/debug" ]

ENTRYPOINT [ "/bin/sh", "-c" ]
```

Listado de código C.1: `Dockerfile` que define una imagen para la ejecución de `rt-tests`.

La imagen base escogida es la edición liviana o *slim* de la versión más reciente de Debian disponible. En este caso, realmente no importaba demasiado cuál elegir, tan solo se necesitan las utilidades mínimas. A continuación, se deben instalar las dependencias necesarias, entre las que figuran `git`, la librería de NUMA (*Non Uniform Memory Access*) usada por `rt-tests`

[1] El contenedor solo ha sido probado con `cyclictest`, `cyclicdeadline` y `deadline_test`, aunque debería funcionar correctamente para el resto de utilidades también.

y los paquetes necesarios para la construcción del código. Una vez hecho esto, se clona el repositorio de `rt-tests` y se cambia el directorio de trabajo a la nueva carpeta. Las distintas versiones de la suite se encuentran identificadas en el repositorio por *tags* o etiquetas, por lo que obtenemos la versión deseada descargando los *commits* de su etiqueta en una nueva rama. En nuestro caso, y como se puede apreciar en el listado, estamos usando la versión 1.8, siendo la 1.9 la más reciente en el momento en el que se realizó el trabajo. El motivo por el que se ha escogido esta versión en vez de la 1.9 es sencillamente que esta última provoca fallos en la compilación para arquitecturas ARM debido a la introducción de una nueva dependencia. Una vez tenemos el código descargado, construimos e instalamos las librerías y los ejecutables con `make`.

Se puede apreciar como, a continuación de esta compilación, se declara un volumen para el contenedor. En Docker, los volúmenes son mecanismos que permiten vincular un directorio del sistema de ficheros del anfitrión con otro del sistema de ficheros del contenedor, de forma que su contenido sea accesible desde dentro del mismo. Esto es necesario para la carpeta `/sys/kernel/debug`, ya que `rt-tests` lee varios de los ficheros que contiene, especialmente `sched_features`, para determinar las propiedades del sistema.

Para terminar, se define `/bin/sh -c` como el punto de entrada del contenedor. Esto quiere decir que, al ejecutarlo, se lanzará este comando y se pasarán como argumentos los que se indiquen en el `docker run`. De esta forma, tenemos la posibilidad de elegir la prueba de `rt-tests` que se desea ejecutar en cada caso.

```
docker run
    -v /sys/kernel/debug:/sys/kernel/debug
    --cap-add SYS_NICE --cap-add IPC_LOCK
    rt-tests "cyclicdeadline -D 10m -t 4"
```

Listado de código C.2: Ejemplo de ejecución de `cyclicdeadline` usando el contenedor.

En el listado C.2, se muestra un ejemplo de ejecución de `cyclicdeadline` usando la imagen de contenedor definida anteriormente. Como se puede apreciar, es necesario usar varios argumentos en la ejecución de `docker run` para hacer que el contenedor se lance correctamente. Estos argumentos son:

- `-v`: Se trata de la conexión del volumen definido dentro del contenedor y que hemos explicado antes, con un directorio del sistema de ficheros del anfitrión. Aunque no es necesario usar el mismo directorio tanto en el anfitrión como en el contenedor, en este caso sí que lo es debido ya que los ficheros deben estar dentro de ese directorio específico.

- `--cap-add`: Con este argumento, podemos indicar algunas *capabilities* o capacidades de Linux que queremos que tenga el contenedor. Estas capacidades permiten realizar diversas operaciones. Para nuestro contenedor, necesitamos dos: `SYS_NICE`, para asignar prioridades a los procesos (valor *nice*); e `IPC_LOCK`, para usar las funciones de reserva de memoria (p. ej., `mlock` o `mlockall`).

Después del nombre de la imagen a ejecutar, que en nuestro caso es `rt-tests`, se indica entre comillas la cadena de texto que se le pasará al punto de entrada del contenedor, es decir, a

`/bin/sh -c`. En el ejemplo del listado, se ejecuta la prueba `cyclicdeadline` con una duración de 10 minutos y un total de 4 hilos.

Bibliografía

[1] W. V. Casteren, "The Waterfall Model and the Agile Methodologies : A comparison by project characteristics," 2017. Publisher: Unpublished.

[2] M. Wolf, *High-Performance Embedded Computing: Applications in Cyber-Physical Systems and Mobile Computing*. Elsevier, 2 ed., Apr. 2014.

[3] E. A. Lee and S. A. Seshia, *Introduction to Embedded Systems: A Cyber-Physical Systems Approach*. MIT Press, 2 ed., Dec. 2016. Google-Books-ID: chPiDQAAQBAJ.

[4] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*. Artech House, 2001. Google-Books-ID: O50vDwAAQBAJ.

[5] B. Randell, "System structure for software fault tolerance," *ACM SIGPLAN Notices*, vol. 10, pp. 437–449, June 1975.

[6] A. Schiper, "Dependable Systems," in *Dependable Systems: Software, Computing, Networks* (D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, J. Kohlas, B. Meyer, and A. Schiper, eds.), vol. 4028, pp. 34–54, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. Series Title: Lecture Notes in Computer Science.

[7] A. A. Amin and K. M. Hasan, "A review of Fault Tolerant Control Systems: Advancements and applications," *Measurement*, vol. 143, pp. 58–68, Sept. 2019.

[8] L. Shan, B. Sangchoolie, P. Folkesson, J. Vinter, E. Schoitsch, and C. Loiseaux, "A Survey on the Application of Safety, Security, and Privacy Standards for Dependable Systems," in *2019 15th European Dependable Computing Conference (EDCC)*, (Naples, Italy), pp. 71–72, IEEE, Sept. 2019.

[9] A. Gambier, "Real-time control systems: a tutorial," in *Proceedings of the 5th Asian Control Conference (IEEE Cat. No.04EX904)*, vol. 2, (Melbourne, Australia), pp. 1024–1031, IEEE, July 2004.

[10] M. Krotofil and D. Gollmann, "Industrial control systems security: What is happening?," in *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, pp. 670–675, July 2013. ISSN: 2378-363X.

[11] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*, (Tucson, AZ, USA), pp. 239–243, IEEE, Dec. 2007.

[12] A. Burns and R. I. Davis, "Mixed Criticality Systems - A Review," 2015.

[13] D. Geer, "Survey: Embedded Linux Ahead of the Pack," *IEEE Distributed Systems Online*, vol. 5, pp. 3–3, Oct. 2004.

[14] J. Henkel and M. Tins, "Munich/MIT Survey: Development of Embedded Linux," Jan. 2004.

[15] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," Tech. Rep. 800-145, National Institute of Standards and Technology, Gaithersburg, USA, Sept. 2011.

[16] F. Hofer, M. A. Sehr, A. Iannopollo, I. Ugalde, A. Sangiovanni-Vincentelli, and B. Russo, "Industrial Control via Application Containers: Migrating from Bare-Metal to IAAS," in *Proceedings of the 11th IEEE International Conference on Cloud Computing Technology and Science (CloudCom)* (J. Chen and L. T. Yang, eds.), (Sydney, Australia), pp. 62–69, IEEE, Dec. 2019. ISSN: 2330-2186.

[17] R. Piggin, "Are industrial control systems ready for the cloud?," *International Journal of Critical Infrastructure Protection*, vol. 9, pp. 38–40, June 2015.

[18] W. Shi, G. Pallis, and Z. Xu, "Edge Computing [Scanning the Issue]," *Proceedings of the IEEE*, vol. 107, pp. 1474–1481, Aug. 2019. Conference Name: Proceedings of the IEEE.

[19] J. Gedeon, J. Heuschkel, L. Wang, and M. Mühlhäuser, "Fog Computing: Current Research and Future Challenges," Mar. 2018.

[20] P. Pop, M. L. Raagaard, M. Gutierrez, and W. Steiner, "Enabling Fog Computing for Industrial Automation Through Time-Sensitive Networking (TSN)," *IEEE Communications Standards Magazine*, vol. 2, pp. 55–61, June 2018. Conference Name: IEEE Communications Standards Magazine.

[21] S. Yi, Z. Hao, Z. Qin, and Q. Li, "Fog Computing: Platform and Applications," in *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pp. 73–78, Nov. 2015.

[22] N. Verba, K.-M. Chao, J. Lewandowski, N. Shah, A. James, and F. Tian, "Modeling industry 4.0 based fog computing environments for application analysis and deployment," *Future Generation Computer Systems*, vol. 91, pp. 48–60, Feb. 2019.

[23] F. Tseng, M. Tsai, C. Tseng, Y. Yang, C. Liu, and L. Chou, "A Lightweight Autoscaling Mechanism for Fog Computing in Industrial Applications," *IEEE Transactions on Industrial Informatics*, vol. 14, pp. 4529–4537, Oct. 2018. Conference Name: IEEE Transactions on Industrial Informatics.

[24] Y. Lu, "Industry 4.0: A survey on technologies, applications and open research issues," *Journal of Industrial Information Integration*, vol. 6, pp. 1–10, June 2017.

[25] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet of Things Journal*, vol. 3, pp. 637–646, Oct. 2016.

[26] Y. Wang, K. Wang, H. Huang, T. Miyazaki, and S. Guo, "Traffic and Computation Co-Offloading With Reinforcement Learning in Fog Computing for Industrial Applications," *IEEE Transactions on Industrial Informatics*, vol. 15, pp. 976–986, Feb. 2019. Conference Name: IEEE Transactions on Industrial Informatics.

[27] W. Z. Khan, E. Ahmed, S. Hakak, I. Yaqoob, and A. Ahmed, "Edge computing: A survey," *Future Generation Computer Systems*, vol. 97, pp. 219–235, Aug. 2019.

[28] A. Stanciu, "Blockchain Based Distributed Control System for Edge Computing," in *2017 21st International Conference on Control Systems and Computer Science (CSCS)*, (Bucharest, Romania), pp. 667–671, IEEE, May 2017.

[29] A. Khan and K. Turowski, "A Perspective on Industry 4.0: From Challenges to Opportunities in Production Systems:," in *Proceedings of the International Conference on Internet of Things and Big Data*, (Rome, Italy), pp. 441–448, SCITEPRESS - Science and and Technology Publications, 2016.

[30] M. A. Farsi and E. Zio, "Industry 4.0: Some Challenges and Opportunities for Reliability Engineering," *International Journal of Reliability, Risk and Safety: Theory and Application*, vol. 2, pp. 23–34, Dec. 2019.

[31] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, pp. 412–421, July 1974.

[32] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP '03, (New York, USA), pp. 164–177, ACM Press, Oct. 2003.

[33] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in Xen," in *Proceedings of the 9th ACM international conference on Embedded software - EMSOFT '11*, (Taipei, Taiwan), pp. 39–48, ACM Press, Oct. 2011.

[34] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My VM is Lighter (and Safer) than your Container," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), pp. 218–233, Association for Computing Machinery, Oct. 2017.

[35] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, (Philadelphia, PA, USA), pp. 171–172, IEEE, Mar. 2015.

[36] A. Randal, "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers," *ACM Computing Surveys*, vol. 53, pp. 1–31, May 2020. arXiv: 1904.12226.

[37] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the Cloud," *Future Generation Computer Systems*, vol. 68, pp. 175–182, Mar. 2017.

[38] M. Cinque, R. D. Corte, A. Eliso, and A. Pecchia, "RT-CASEs: Container-Based Virtualization for Temporally Separated Mixed-Criticality Task Sets," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)* (S. Quinton, ed.), vol. 133 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 5:1–5:22, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019. ISSN: 1868-8969.

[39] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-Time Containers: A Survey," in *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)* (A. Cervin and Y. Yang, eds.), vol. 80 of *OpenAccess Series in Informatics (OASIcs)*, (Dagstuhl, Germany), pp. 7:1–7:9, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2020. ISSN: 2190-6807.

[40] W. Wang, P. Mishra, and S. Ranka, "Introduction," in *Dynamic Reconfiguration in Real-Time Systems: Energy, Performance, and Thermal Perspectives*, pp. 1–13, New York, NY: Springer New York, 2013.

[41] E. Rotenberg and A. Anantaraman, "Chapter 4 - Architecture of Embedded Microprocessors," in *Multiprocessor Systems-on-Chips* (A. A. Jerraya and W. Wolf, eds.), Systems on Silicon, pp. 81–112, San Francisco: Morgan Kaufmann, Jan. 2005.

[42] J. Anderson, V. Bud, and U. Devi, "An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, (Palma de Mallorca, Balearic Islands, Spain), pp. 199–208, IEEE, 2005.

[43] A. Burns and A. J. Wellings, *Real-time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley, 4 ed., 2009. Google-Books-ID: XO9dPgAACAAJ.

[44] A. Burns, "Scheduling hard real-time systems: a review," *Software Engineering Journal*, vol. 6, no. 3, p. 116, 1991.

[45] J. Abella, C. Hernandez, E. Quinones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, (Siegen, Germany), pp. 1–10, IEEE, June 2015.

[46] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, vol. 20, pp. 46–61, Jan. 1973.

[47] J. P. Lehoczky, L. Sha, J. K. Strosnider, and H. Tokuda, "Fixed Priority Scheduling Theory for Hard Real-Time Systems," in *Foundations of Real-Time Computing: Scheduling and Resource Management* (A. M. Tilborg and G. M. Koob, eds.), pp. 1–30, Boston, MA: Springer US, 1991.

[48] N. Fisher and S. Baruah, "Rate-Monotonic Scheduling," in *Encyclopedia of Algorithms* (M.-Y. Kao, ed.), pp. 751–754, Boston, MA: Springer US, 2008.

[49] N. Audsley, A. Burns, M. Richardson, and A. Wellings, "Hard Real-Time Scheduling: The Deadline-Monotonic Approach," *IFAC Proceedings Volumes*, vol. 24, pp. 127–132, May 1991.

[50] F. Zhang and A. Burns, "Schedulability Analysis for Real-Time Systems with EDF Scheduling," *IEEE Transactions on Computers*, vol. 58, pp. 1250–1258, Sept. 2009.

[51] Stewart and Khosla, "Real-time scheduling of dynamically reconfigurable systems," in *IEEE International Conference on Systems Engineering*, (Fairborn, OH, USA), pp. 139–142, IEEE, 1991.

[52] B. Sprunt, L. Sha, and J. Lehoczky, "Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System," tech. rep., Defense Technical Information Center, Fort Belvoir, VA, Apr. 1989.

[53] P. Hambarde, R. Varma, and S. Jha, "The Survey of Real Time Operating System: RTOS," in *2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies*, (Nagpur, India), pp. 34–39, IEEE, Jan. 2014.

[54] "Deadline Task Scheduling." `https://www.kernel.org/doc/html/latest/scheduler/sched-deadline.html`. Accessed: 2020-11-24.

[55] F. Reghenzani, G. Massari, and W. Fornaciari, "The Real-Time Linux Kernel: A Survey on PREEMPT_rt," *ACM Computing Surveys*, vol. 52, pp. 1–36, Feb. 2019.

[56] R. T. Fielding, *Architectural styles and the design of network-based software architectures*. phd, University of California, Irvine, 2000. AAI9980887 ISBN-10: 0599871180.