Escuela Técnica Superior de Ingenieros
Industriales y de Telecomunicación
UNIVERSIDAD DE CANTABRIA

Trabajo Fin de Máster

Plataforma de simulación multinivel y análisis de prestaciones para sistemas robóticos basados en ROS

(ROS-based multilevel robotic systems simulation and performance analysis platform)

Para acceder al Título de

Máster Universitario en
Ingeniería de Telecomunicación

Autor: Javier Merino Calleja

Julio - 2021

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACION

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MASTER

Realizado por: Javier Merino Calleja
Director del TFM: Eugenio Villar Bonet
Título: "Plataforma de simulación multinivel y análisis de prestaciones para sistemas robóticos basados en ROS"
Title: "ROS-based multilevel robotic system simulation and performance analysis platform"

Presentado a examen el día:

para acceder al Título de

MASTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del Tribunal:
Presidente (Apellidos, Nombre): Jose María Zamanillo Sainz de la Maza
Secretario (Apellidos, Nombre): Víctor Fernández Solórzano
Vocal (Apellidos, Nombre): Mauro Lomer Barbosa

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFM (sólo si es distinto del Secretario)
Vº Bº del Subdirector	Trabajo Fin de Máster Nº (a asignar por Secretaría)

Resumen

La detección de errores en el diseño de un sistema debe producirse cuanto antes dentro del flujo del desarrollo, y poder realizarse en todas las etapas del mismo. La simulación sobre un modelo virtual se ha convertido en una técnica fundamental dentro del proceso de diseño de cualquier sistema electrónico, ya que permite verificar el correcto funcionamiento del sistema en las diferentes etapas del desarrollo, y en caso de detectar errores, corregirlos y probarlo de nuevo con gran agilidad. Para ello, el Grupo de Sistemas Embebidos de la Universidad de Cantabria ha desarrollado la herramienta Virtual Parallel Platform for Performance Estimation (VIPPE), una infraestructura para la simulación y estimación de rendimiento de sistemas embebidos complejos y heterogéneos, que permite realizar todas estas tareas dentro de un ecosistema de desarrollo único.

Dentro del ámbito de la electrónica, la robótica es claro ejemplo de un campo que ha sufrido un desarrollo exponencial en los últimos años, y en el cual resulta muy apropiado la aplicación de las técnicas de simulación. De entre todas las plataformas actuales de desarrollo de código para robots, Robot Operating System (ROS) se ha establecido como una de las más importantes, soportando varios lenguajes de programación y siendo de código abierto y uso libre.

En este trabajo se aborda la integración del entorno ROS dentro del simulador VIPPE para crear una infraestructura sobre la que poder simular y obtener prestaciones de servicios complejos que cuenten entre otros con componentes robóticos. Adicionalmente, esta plataforma deberá ser capaz de simular modelos de componentes a diferentes niveles de abstracción que den lugar a simulaciones multinivel, de gran utilidad en el desarrollo de sistemas.

Abstract

Early error detection is fundamental during the design process of a system. An error revealed when the system has already been finalized would imply a high cost, resulting in a slow, iterative process. For all these reasons, simulation over virtual models has become a core technique within the design process of any electronic system. It allows verifying the correctness of the system during different stages of development, achieving design error detection for further correction and testing, resulting in a more efficient process. For this purpose, the Embedded Systems Group of the University of Cantabria has developed the Virtual Parallel Platform for Performance Estimation (VIPPE) tool, an infrastructure for simulation and performance estimation of complex and heterogeneous embedded systems, which allows all these tasks to be carried out inside a unique development ecosystem.

Within the electronics field, robotics is a clear example of an application field that has undergone exponential development in recent years, and in which the usage of simulation techniques is very appropriate. Among all current robotic code developing platforms, the open-source Robot Operating System (ROS) has established itself as one of the most widely used, supporting various programming languages.

This work faces the integration of the ROS environment within the VIPPE simulator to create an infrastructure on which to simulate and obtain performance metrics of complex services that include robotic components, among others. Additionally, this platform should be able to simulate component models at different levels of abstraction, resulting on multi-level simulations which are of great interest during the development process of a system or service.

Agradecimientos

En estas líneas me gustaría agradecer a todas aquellas personas que me han acompañado a lo largo de estos años, y que han facilitado de uno u otro modo la realización de este trabajo.

En primer lugar me gustaría agradecer a Eugenio Villar darme la oportunidad de trabajar durante estos casi tres años en investigación dentro del Grupo de Ingeniería de Sistemas Embebidos, formando parte de los proyectos MegaMart2 y COMP4DRONES, siendo éste último el origen de este trabajo.

Seguidamente, nada de todo esto hubiera sido posible sin la ayuda constante de Héctor Posadas, siempre dispuesto a echarme una mano cuando lo necesitaba, y del cual me llevo una gran cantidad de cosas aprendidas. Muchísimas gracias por tu paciencia y apoyo.

A todos mis compañeros de trabajo, desde el momento en el que entré hasta el día de hoy, especialmente a Ángel, Simon, Raquel y Diego. Gracias por los grandes ratos y las buenas pausas del café, se echarán en falta.

A Fernando, mi compañero de fatigas en el máster y en el trabajo sin el cual todo esto hubiera sido muchísimo más complicado. He tenido mucha suerte de tenerte ahí siempre que lo necesitaba, en los buenos momentos y en lo malos. Muchas gracias por todo amigo.

Finalmente, a mi familia y amigos, por estar a mi lado y apoyarme todo este tiempo.

Muchas gracias a todos

Índice general

Índice de figuras

Índice de tablas

Lista de acrónimos

CPS Cyber-Physical Systems.

DSE Design Space Exploration.

ISS Instruction Set Simulator.

MPSoC Multiprocessor System on a Chip.

ROS Robot Operating System.

RPC Remote Procedure Call.

RTL Register Transfer Level.

RTOS Real Time Operating System.

SoC System on a Chip.

UAV Unmanned Aerial Vehicle.

VIPPE Virtual Parallel Platform for Performance Estimation.

CAPÍTULO

1 Introducción

Dentro del flujo de diseño de cualquier sistema electrónico, verificar el correcto funcionamiento del mismo antes de llevarlo a la realidad es una parte fundamental. En el pasado, esto sólo era posible creando un prototipo físico del sistema y realizando pruebas sobre él, lo cual resulta un proceso lento, costoso y que potencialmente requería repetir gran parte del diseño en caso de detectarse errores. Con el avance de la tecnología, resulta posible realizar una verificación del sistema realizando simulaciones del mismo antes de llevarlo a la realidad. Por ello, surgieron técnicas para incorporar la verificación y validación a lo largo del proceso de diseño. Entre ellas, destaca la metodología de diseño en V, ampliamente empleada en el desarrollo de sistemas y mostrado en la Figura 1.1. Este método se basa en realizar una separación de los diferentes niveles de abstracción del sistema. La parte izquierda de la V establece la identificación de los requerimientos del sistema, su arquitectura, diseño y finalmente en un nivel más bajo de abstracción la implementación. En el lado derecho se representa la integración de los subsistemas que forman el todo y su validación. En cada nivel de abstracción se realiza una verificación y/o validación entre el lado derecho e izquierdo de la V, para que en caso de detectarse errores estos puedan ser corregidos lo antes posible. Es por tanto, un método iterativo de diseño, que asegura que el diseño final tenga una robustez y prestaciones de acuerdo con lo especificado.

Para realizar la verificación, uno de los métodos más populares empleados en la actualidad es la simulación sobre un modelo virtual del sistema desarrollado en software. Estas simulaciones deben ser rápidas, precisas y lo suficientemente sencillas para el desarrollador de cara a no enlentecer un proceso de diseño que, generalmente, necesita ser iterativo

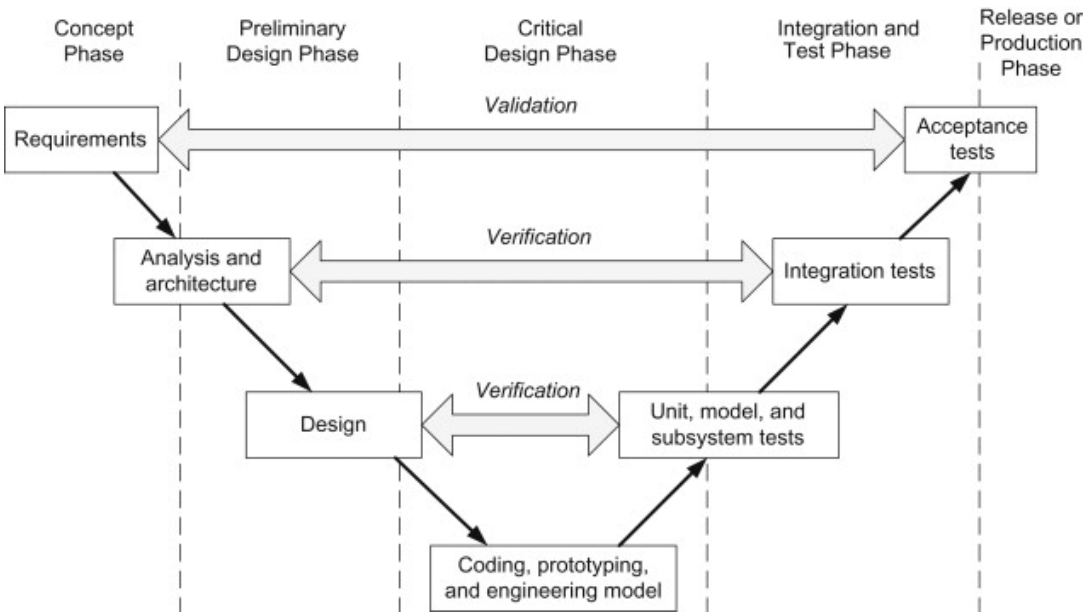

Figura 1.1: Modelo de diseño en V

hasta obtener el resultado adecuado. Además, resulta posible determinar diferentes niveles de simulación en función de la complejidad del modelo, desde lo puramente funcional mediante un modelo muy simple hasta simulaciones muy complejas de sistemas ciberfísicos (CPS). Para ello queda del lado del desarrollador escoger qué niveles se adecúan a sus necesidades teniendo en cuenta el compromiso entre velocidad de simulación y la precisión de los resultados obtenidos.

Todo lo mencionado se puede extrapolar a un nivel superior de abstracción donde se encuentran los servicios. Un servicio puede estar compuesto por un único sistema, o bien por un conjunto muy complejo de ellos que interactúan entre sí mediante distintos mecanismos de comunicación. Por tanto, resulta muy conveniente que los modelos y las herramientas de simulación puedan ser escalables para realizar la verificación de estos servicios complejos.

En el contexto de desarrollo actual, han surgido numerosos servicios que hacen uso de sistemas robóticos, como drones o brazos articulados, empleándose en ámbitos tan dispares como el industrial, comercial o de ocio. Como elemento común de todos estos sistemas robóticos destaca el empleo de software para la programación de su funcionalidad y control, y por tanto dicho software puede ser empleado de manera directa como modelo de simulación. En los últimos años han surgido diferentes plataformas que permiten la simulación y despliegue de sistemas robóticos de manera rápida y distribuida, tales como Robot Operating System (ROS), Microsoft Robotics Studio, Orca u Orocos. Éstas habitualmente

proporcionan una serie de utilidades que facilitan la programación de sistemas robóticos tales como despliegue distribuido, comunicación en red, manejo avanzado de tiempo, o herramientas mas específicas de la aplicación a desarrollar como tratamiento de imágenes o sensórica.

De entre ellas, ROS destaca por ser la más utilizada en el ámbito de la programación de sistemas robóticos. Esto se debe principalmente a que es de código abierto bajo licencia BSD, y que está enfocada a la reutilización de código, facilitando el intercambio de software entre desarrolladores a través de una gran comunidad de usuarios. Además, soporta de manera oficial lenguajes de programación tan utilizados como C++ y Python, lo que hace que sea fácilmente asumible por los desarrolladores.

1.1. Motivación

Introducido el contexto del trabajo, cabe destacar que existe una carencia de herramientas que permitan al desarrollador realizar simulaciones de manera rápida y eficiente de componentes descritos en ROS. Es cierto que ROS soporta el empleo de un tiempo emulado en lugar del tiempo real de la plataforma para, por ejemplo, acelerar la simulación de un sistema. Sin embargo, no es posible realizar simulaciones más complejas donde se proporcionen métricas de estimación de prestaciones, como pueden ser el número de ciclos, instrucciones y/o energía que requiere la ejecución de una aplicación sobre una plataforma hardware específica.

El Grupo de de Sistemas Embebidos (GESE), dentro del Grupo de Ingeniería Microelectrónica (GIM) de la Universidad de Cantabria ha desarrollado la herramienta Virtual Parallel Platform for Performance Estimation (VIPPE), una infraestructura para la simulación y estimación de rendimiento de sistemas embebidos multiprocesador (MPSoC) complejos y heterogéneos. VIPPE permite simular aplicaciones sobre un modelo de la plataforma hardware de destino deseada y en función de cómo las diferentes partes de la funcionalidad de la aplicación estén mapeadas sobre la plataforma. De esta manera, es capaz de proporcionar estimaciones precisas sobre tiempo simulado, ciclos, instrucciones, energía consumida, o datos relativos a las cachés del sistema, si las hubiera.

El presente Trabajo de Fin de Máster se ha realizado dentro del grupo de investigación GESE/GIM del Departamento de Tecnología Electrónica e Ingeniería de Sistemas y Automática (TEISA) como parte del proyecto europeo COMP4DRONES. Éste es un proyecto ECSEL liderado por la empresa Indra y formado por un consorcio de 50 miembros, que busca crear un entorno de trabajo para facilitar el desarrollo de tecnologías de drones inteligentes y

autónomos [1]. COMP4DRONES se centra en la creación de un ecosistema de desarrollo que permita integrar en el proceso de diseño los distintos niveles de abstracción, desde la funcionalidad genérica hasta el componente electrónico, y de forma modular, de manera que sea altamente personalizable en función de las necesidades de la aplicación. Por su interés como campos de aplicación de servicios basados en drones inteligentes, se han desarrollado diferentes casos de uso relacionados con el transporte, agricultura, revisión de infraestructuras, logística o paquetería, entre otros.

Al estar centrado el proyecto en el despliegue de servicios basados en drones, resulta muy apropiado el empleo de ROS como plataforma para el desarrollo de código y simulación, debido a dos motivos principales:

- Un dron es un sistema robótico volador, por lo que encaja de manera orgánica en la topología de ROS basada en nodos distribuidos. Además, es de por sí es un elemento funcional del servicio (realiza una acción física) y computacional, al contar con un hardware que ejecuta su funcionalidad.

- Existe una gran cantidad de código ya existente reutilizable para la programación de la funcionalidad de diferentes tipos de drones descrito en ROS, y debido a la amplia comunidad que lo respalda este sigue creciendo día a día.

1.2. Objetivos

El principal objetivo de este trabajo es diseñar e implementar una plataforma para la simulación y análisis de prestaciones de sistemas y servicios que incluyan componentes robóticos ROS. Para lograr dicho objetivo, será necesario realizar las adaptaciones necesarias en la herramienta VIPPE para que integre la infraestructura ROS. Como condición, será necesario que dichas modificaciones no perjudiquen la precisión y velocidad de VIPPE a la hora de simular aplicaciones. Una vez establecida esta infraestructura, se establecerán diferentes niveles de abstracción y se creará un entorno de simulación, particularizando para drones, y así realizar simulaciones multinivel de este tipo de sistemas.

Para testar el correcto funcionamiento del sistema, se ha desarrollado un caso de uso de un servicio de paquetería en el cual el dron entrega un pedido realizado a través de internet en la posición GPS especificada, siguiendo una serie de coordenadas. Para el dron se ha escogido la versión de autopiloto para vehículos aéreos no tripulados (UAV) de ArduPilot, ArduCopter [2], el cual está diseñado para ser empleado a nivel de usuario, pero al ser de código abierto resulta también apropiado para su uso en investigación.

1.3. Estructura del documento

En esta sección se detalla la estructura del documento, así como una breve descripción de cada uno de los apartados que lo forman

- **Capítulo 1. Introducción**
  El capítulo actual. Se enmarca el trabajo dentro del contexto de simulación de servicios basados en sistemas robóticos y el uso de ROS como plataforma de desarrollo. También se introduce VIPPE como infraestructura de simulación y análisis de prestaciones de aplicaciones diseñadas para sistemas embebidos. De estos dos actores principales surge la motivación del trabajo y los objetivos que se pretenden alcanzar.

- **Capítulo 2. Estado del arte**
  En este apartado se realiza un repaso de la literatura de los principales pilares que sustentan este trabajo, VIPPE y ROS. Adicionalmente, se describen algunos aspectos acerca del resto de elementos que dan forma al trabajo, como es el caso de las simulaciones multinivel.

- **Capítulo 3. Integración de ROS en VIPPE**
  El capítulo 3 aborda el tema principal del trabajo, como es la adaptación de VIPPE para soportar simulaciones de componentes descritos en ROS. Entre los aspectos más relevantes a considerar está la sincronización entre los dos espacios de simulación, la anotación del código ROS en el proceso de compilación y la simulación en tiempo real.

- **Capítulo 4. Modelos robóticos: Particularización para drones**
  Debido al amplio espectro de posibles sistemas robóticos, en este capítulo se realiza una particularización para el caso de los drones. De esta manera, se detallan diferentes niveles de simulación en función de la complejidad de los modelos de dron empleados, desde un dron puramente funcional hasta un modelo completo y real como es ArduCopter. Finalmente, se describe la estructura de elementos para realizar las simulaciones multinivel.

- **Capítulo 5. Ejemplo de aplicación: Servicios basados en drones**
  Para comprobar la correcta ejecución de los programas ROS bajo la simulación de VIPPE, y los elementos incluidos para la simulación multinivel, se ha desarrollado un caso de uso como banco de pruebas de un servicio que hace uso de drones para el envío de paquetería. Sobre este ejemplo se realizaron diferentes experimentos para emular

el flujo de diseño de un servicio complejo de estas características, proporcionando métricas relevantes para el mismo.

- **Capítulo 6. Conclusiones**
  El último capítulo describe las conclusiones más importantes del trabajo realizado, así como las contribuciones científicas derivadas del mismo y las líneas futuras de mejora.

CAPÍTULO

2 Estado del arte

Con el avance de la tecnología, los sistemas electrónicos son cada vez más complejos con el objetivo de proporcionar un mayor número de servicios en un espacio muy reducido. Limitaciones como el tamaño, peso y consumo energético, unidas a la necesidad de ejecutar tareas con un creciente coste computacional, hacen que la integración de estos sistemas sea un aspecto crucial. En la actualidad, gracias a la enorme cantidad de transistores disponibles por circuito integrado, resulta posible dicha integración de procesadores y componentes hardware sobre el mismo circuito dando lugar a los sistemas en chip con uno (SoC) o múltiples (MPSoC) procesadores. Esta heterogeneidad en su composición, incluyendo hardware específico diseñado a medida con software embebido, hacen que el diseño hardware-software estén íntimamente relacionados para poder obtener las mejores prestaciones del sistema. A modo ilustrativo, en el extremo contrario se encontraría un software genérico que se ejecute sobre una unidad de cómputo de propósito general, como un ordenador personal que, aunque destaca por su gran flexibilidad, no alcanza el rendimiento de los sistemas embebidos.

A continuación se realizará un breve estudio de los diferentes elementos que componen este proyecto.

2.1. VIPPE

Tal y como se ha presentado en la introducción, la simulación se ha convertido en una técnica indispensable hoy en día para poder llevar a cabo el diseño e implementación de cualquier sistema electrónico o servicio. Permite al desarrollador conocer en un

estado muy temprano del proceso de diseño, prácticamente después de haberse fijado las especificaciones, cuál será el comportamiento aproximado del sistema una vez finalizado.

Las peculiaridades descritas de los sistemas embebidos hacen más relevante aún el empleo de simulaciones durante la etapa de diseño de este tipo de sistemas, permitiendo un desarrollo en paralelo de la parte hardware y software. En un primer momento puede emplearse un modelo simple, funcional y estructural del hardware sobre el que continuar el diseño del software. A medida que el proceso de diseño avance, estos modelos pueden ir completándose para aumentar su complejidad y acercarlos a una versión más cercana a la realidad. Además, el empleo de simulaciones permite, aparte de verificar el comportamiento del sistema, optimizar su diseño empleando técnicas de Exploración del Espacio de Diseño (DSE)[3], evaluando todas las combinaciones posibles y así obtener la que ofrece unas mejores prestaciones.

2.1.1. Tipos de simulación

Entre las diferentes técnicas de simulación disponibles para el diseñador, existe un compromiso entre la precisión alcanzada y la velocidad de simulación. Entre los métodos más empleados en la actualidad destacan:

- **Descripción a nivel transferencia de registros (RTL)**: A partir de la definición de componentes mediante lenguajes de descripción hardware como VHDL o Verilog, es posible obtener de manera automática empleando herramientas de diseño el circuito electrónico compuesto por puertas lógicas. Dichas herramientas son capaces de simular los circuitos, obteniendo una elevada precisión. Sin embargo, los tiempos de simulación en circuitos complejos como los MPSoC son tan elevados que no resulta viable su empleo, sobre todo en las primeras etapas del proceso de diseño. Este tipo de simulaciones se suele emplear una vez el diseño está cercano a ser concluido para verificar de la manera más exacta posible su correcto funcionamiento, antes de ser enviado para fabricación.

- **Simulador de conjunto de instrucciones (ISS)**: Estos simuladores surgen de la necesidad de obtener simulaciones más veloces que las RTL. Para ello, incorporan el conjunto de instrucciones de un procesador determinado y así poder ejecutar los códigos binarios cross-compilados para ese procesador. Además, incorporan información funcional acerca de características propias del procesador, como la topología de cachés o el número de núcleos. Una de las técnicas más utilizadas es la combinación de ISS y RTL para la simulación de plataformas heterogéneas [4], como se muestra

en la Figura 2.1. A pesar de obtener mejores tiempos que en el caso anterior, sigue resultando demasiado lento en comparación con el tiempo de ejecución real de la aplicación. Adicionalmente, limitan enormemente la posibilidad de explorar el espacio de diseño al tener una dependencia directa con el hardware empleado.

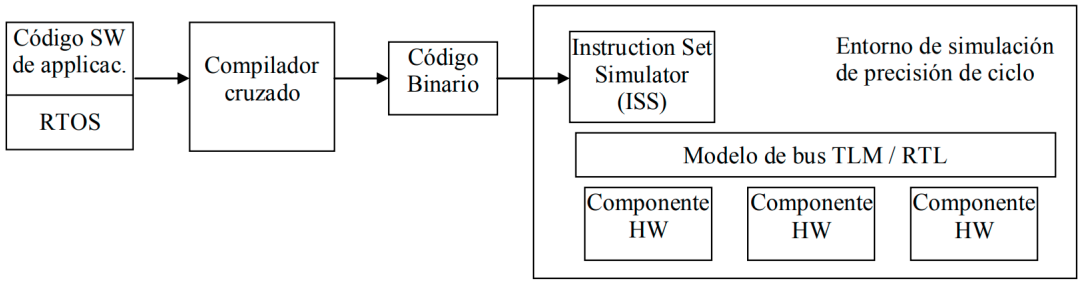

Figura 2.1: Esquema de simulación ISS+RTL [3]

- **Traducción binaria:** En un nivel superior de abstracción y velocidad se encuentran las técnicas de simulación basadas en la traducción de los códigos binarios generados para la plataforma destino (*target*) al procesador nativo que ejecuta la simulación (*host*), de forma dinámica durante la ejecución [5]. Esto evita la necesidad de disponer de un modelo virtual de la plataforma, por lo que la simulación se realiza de manera más rápida que en los casos anteriores. En contrapartida, al no incorporar la información de plataforma, no proporciona estimación de rendimiento del código sobre el hardware, información que resulta muy relevante durante el proceso de diseño. Por ello, suelen emplearse para testar la funcionalidad del código. Un ejemplo de este tipo de simuladores es OVPSim [6].

- **Simulación basada en trazas:** Esta técnica introduce en los bloques básicos (secuencias de instrucciones sin ningún tipo de salto, salvo por la entrada y salida) del código software trazas de información relativas al tiempo que le llevaría al procesador *target* ejecutar ese bloque [7]. De esta manera se obtienen simulaciones relativamente rápidas y precisas. Como contrapartida, para dar sentido a toda la información generada en las trazas, éstas deben ser analizadas posteriormente a la simulación.

- **Simulación nativa:** La última de las técnicas de simulación presentadas es uno de los métodos más empleados en la actualidad por los buenos resultados que ofrece en cuanto a tiempo de simulación, con una precisión suficiente. La simulación nativa se basa en realizar una anotación e instrumentación de los bloques básicos directamente sobre el código software o tras la compilación cruzada, en función de un modelo

virtual de la plataforma hardware a simular [8]. La descripción de las plataformas virtuales resulta un proceso sencillo y rápido de implementar en caso de requerirse una plataforma personalizada, aunque es habitual disponer de una colección de ellas para poder testar diferentes configuraciones. Este es el tipo de simulación sobre el que se fundamenta VIPPE.

2.1.2. Descripción de la herramienta

VIPPE es una plataforma de simulación y análisis de prestaciones enfocada en el co-diseño HW-SW enmarcada dentro de la filosofía de simulación nativa. Recibe como entradas el código fuente escrito en C/C++ y el modelo virtual de plataforma hardware, donde se describen los componentes físicos del sistema, sus interconexiones y el mapeo del software en dicho hardware. Con ello, se obtiene una información muy útil a partir de los resultados de simulación que permiten al desarrollador retroalimentar su diseño para realizar ajustes tanto en la parte hardware como en el código software, tal y como se observa en la Figura 2.2:

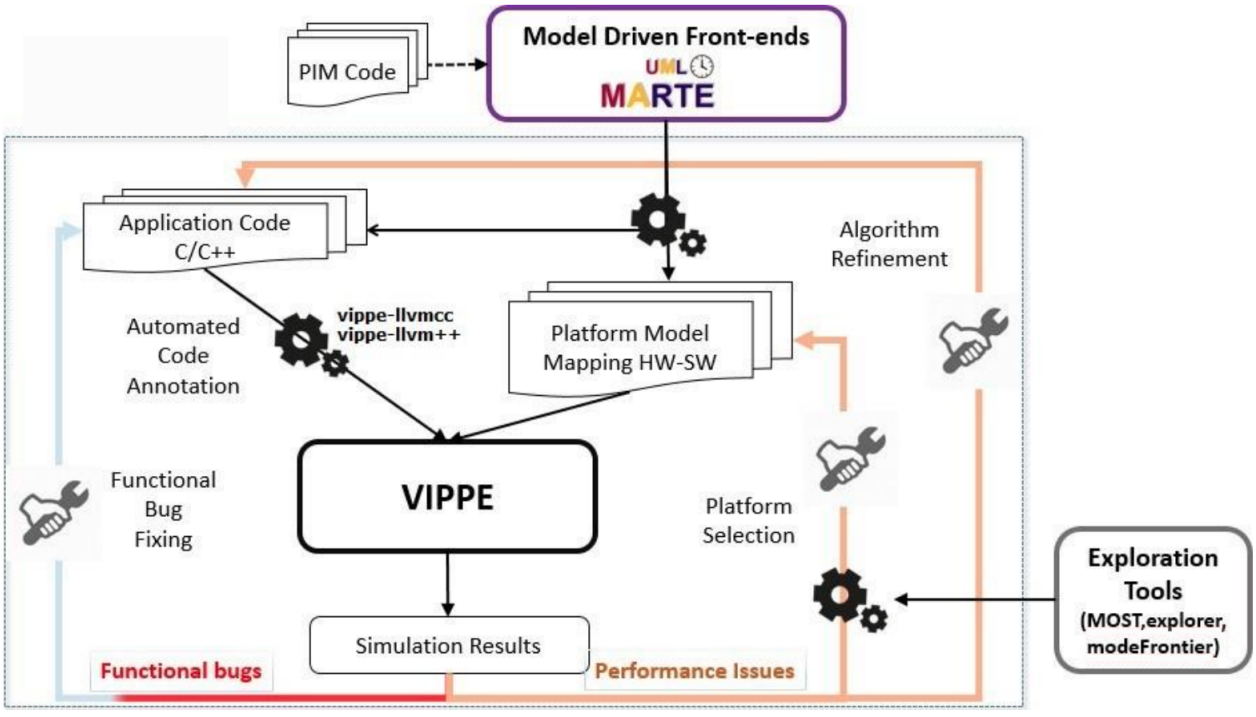

Figura 2.2: Esquema de la plataforma de diseño HW/SW VIPPE [9]

Su funcionamiento se basa en el empleo conjunto de dos herramientas software: el compilador y el simulador.

2.1.2.1. Compilador

En el proceso de compilación, se realiza una anotación del código fuente con instrucciones propias de VIPPE para realizar las métricas de rendimiento. VIPPE hace uso de la infraestructura LLVM [10], y más concretamente de su compilador Clang [11] para obtener los códigos intermedios (.bc) durante la compilación, detectar los bloques básicos, y ahí insertar las líneas de código necesarias. Una vez instrumentado, el código se termina de compilar para obtener el código objeto final (.o). Éste proceso viene ilustrado en la Figura 2.3: El compilador de VIPPE para C++ y C se denomina *vippe-g++* y *vippe-gcc*, respectivamente.

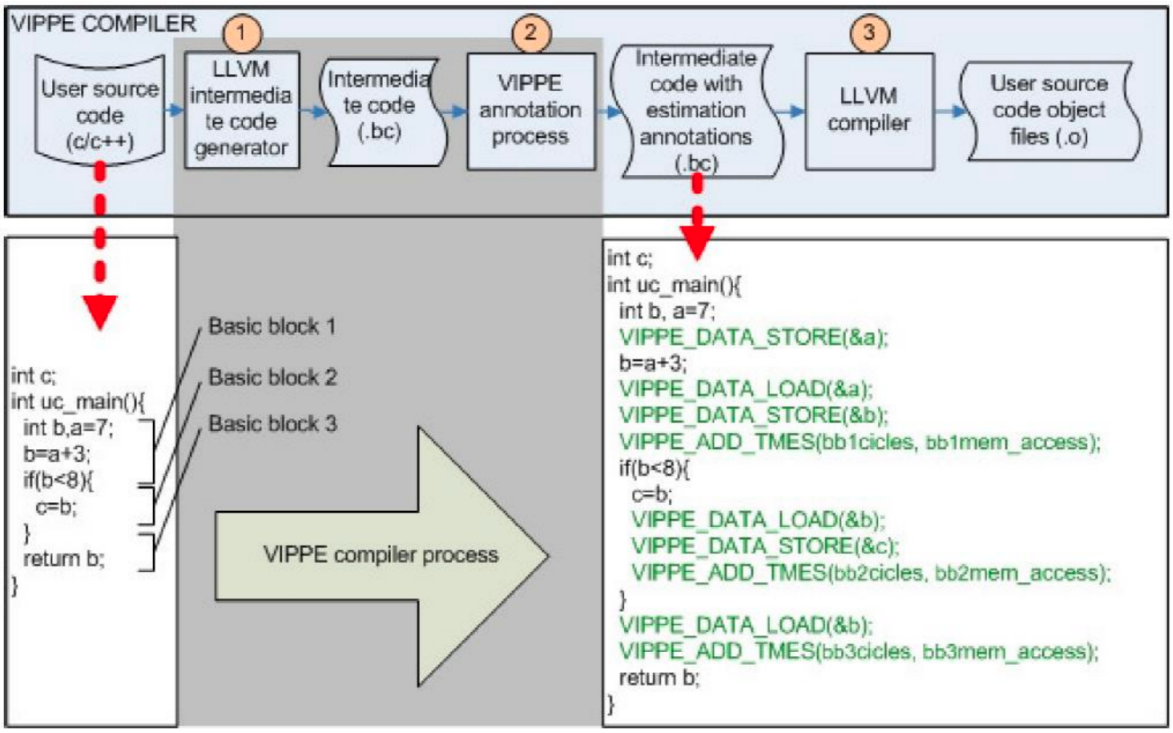

Figura 2.3: Proceso de compilación de VIPPE [12]

2.1.2.2. Simulador

Una vez compilados, los códigos se ejecutan sobre la plataforma local mediante una serie de librerías de simulación. La implementación de la plataforma virtual mapea cada hilo *target* en un hilo de *host*. El sistema operativo de tiempo real (RTOS) *target* se emula mediante un hilo de sistema adicional que ejecuta el kernel de simulación, y es el encargado de realizar la planificación de ejecución de las tareas, creación de procesos, sincronización... La comunicación entre los hilos y el kernel se realiza por medio de memoria compartida, tal y como muestra la Figura 2.4. Para aumentar el paralelismo entre los hilos, la sincronización

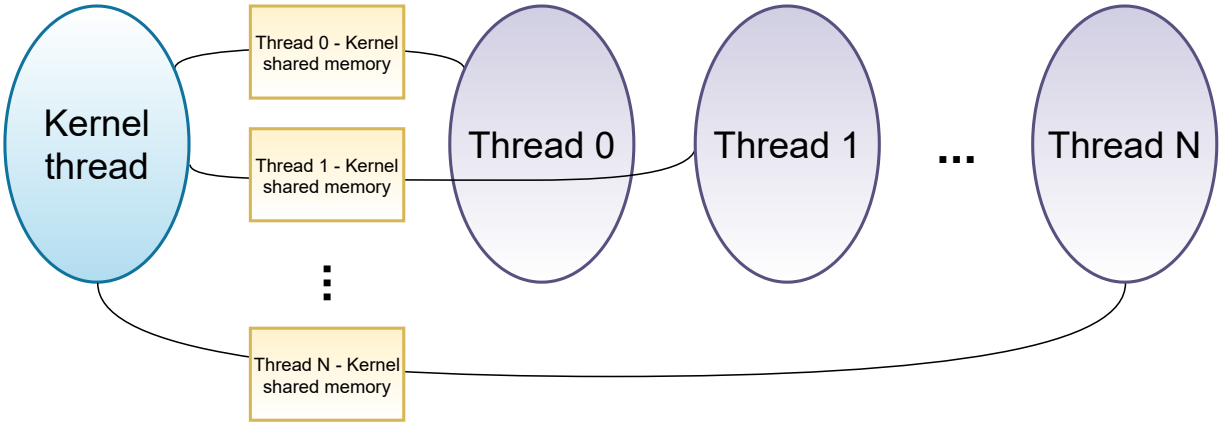

Figura 2.4: Estructura del simulador VIPPE [8]

entre tareas concurrentes se realiza únicamente durante las operaciones de lectura de variables compartidas, lo que consigue evitar bloqueos innecesarios y aumentar la velocidad de simulación.

2.2. ROS

La creación de software de propósito general para el control de sistemas robóticos tiene unas implicaciones en materia de seguridad, robustez y fiabilidad que hacen que esta no sea una tarea en absoluto trivial. ROS nace con el propósito de facilitar el desarrollo de software robótico, proveyendo un entorno de trabajo flexible formado por librerías, servicios y herramientas propias. Está diseñado para ser lo más modular y distribuido posible, de manera que el usuario tiene la opción de integrar en su aplicación código ya existente junto con programas creados *ad-hoc*.

Fue creado en la Universidad de Stanford en el año 2007, y un año después fue trasladado al laboratorio de investigación robótica *Willow Garage*, momento a partir del cual se comenzaron a asentar las bases de lo que sería uno de los proyectos de desarrollo robótico de código abierto más importantes del mundo. Una de las mayores fortalezas de ROS, aparte de su robusta infraestructura, es la gran comunidad de usuarios que lo usa, la cual sigue creciendo año a año tal y como se muestra en la Figura 2.5. De esta red ha surgido un ecosistema de código reutilizable formado por más de 3000 paquetes, implementando funcionalidades que van desde pruebas de concepto de nuevos algoritmos hasta drivers con altos estándares de calidad para uso industrial [13].

Como parte de su flexibilidad, ROS permite desarrollar código en diferentes lenguajes de programación, como C++, Python o Lisp, y ser ejecutado sobre diferentes sistemas operativos

tales como Ubuntu, MacOs y Windows.

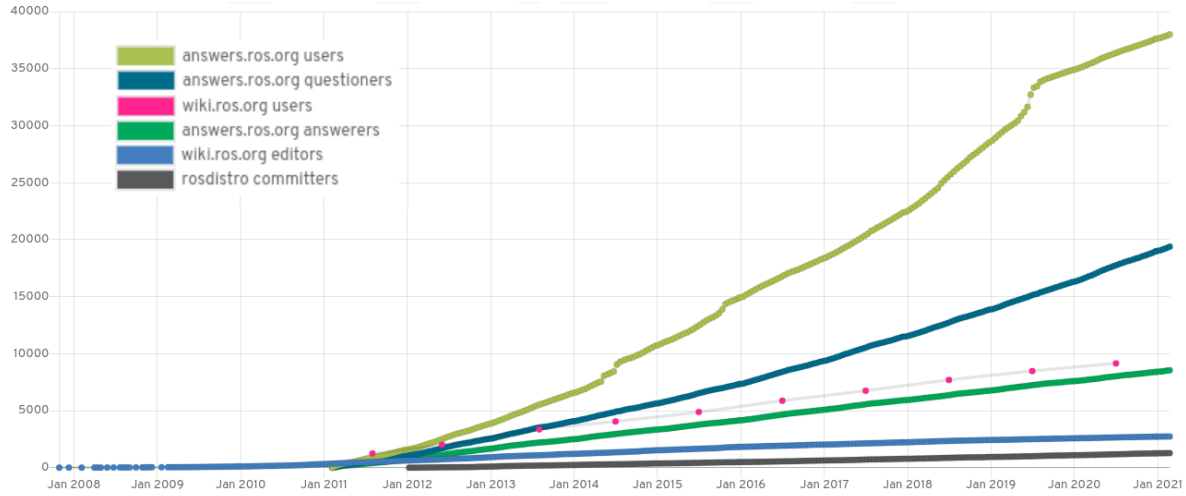

Figura 2.5: Evolución del número de usuarios de ROS

2.2.1. Arquitectura ROS

Como se ha mencionado, ROS proporciona una infraestructura base sobre la que desplegar procesos denominados nodos. Estos nodos se comunican entre sí mediante mensajes a través de temas (*topics*), realizando llamadas a servicios remotos y/o proveyendo dichos servicios a otros nodos. Todo esto es posible gracias al proceso ROS Master, el cual se encarga de mantener un registro de los nodos activos, establecer las conexiones extremo a extremo entre nodos y proporcionar los mecanismos necesarios para las comunicaciones. Esta arquitectura se muestra de manera simplificada en la Figura 2.6. A continuación se va a

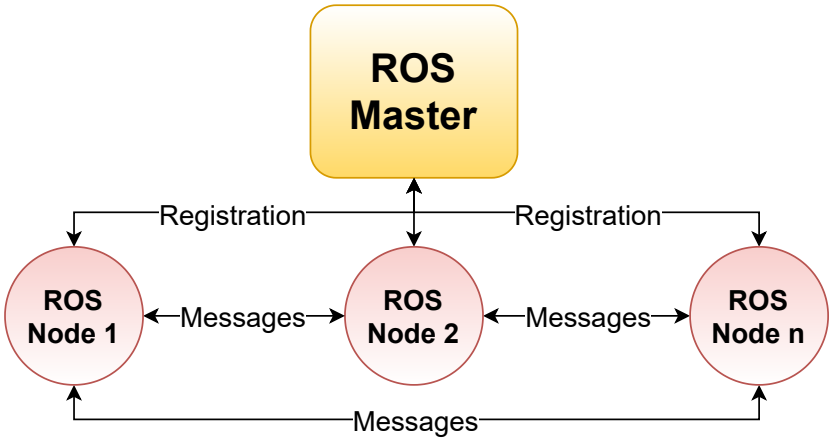

Figura 2.6: Arquitectura de nodos ROS

analizar con mayor detalle los elementos más importantes que forman ROS.

2.2.1.1. Nodos

Un nodo es un programa o proceso que realiza un determinado cómputo. El funcionamiento de una estructura constituida en nodos se basa en repartir la carga computacional del programa de forma distribuida entre diferentes elementos hardware conectados entre sí a través de la red. De esta manera también se favorece la reusabilidad y se reduce considerablemente el esfuerzo necesario para desarrollar la funcionalidad deseada, siendo posible dividir el problema general en problemas más sencillos que son más fácilmente abordables. Además, se aumenta la robustez de la aplicación, obteniendo una mayor tolerancia a fallos puntuales al estar estos aislados en cada nodo.

Dentro del paradigma de sistemas robóticos, la arquitectura de nodos encaja de forma idónea. Por ejemplo, en un robot de tipo rover puede estar formado por varios nodos ROS interactuando entre sí, donde un nodo se encarga de recoger los datos de sensores de distancia, otro controla la velocidad de las ruedas, otro envía imágenes de vídeo a un nodo remoto que las procesa...

Los nodos se registran en la red mediante un nombre que los identifica dentro del sistema. Este identificador debe ser único para evitar conflictos con otros nodos. Para el desarrollo de nodos, ROS proporciona las librerías *roscpp* y *rospy* para C++ y Python, respectivamente.

2.2.1.2. Topics

Los temas o *topics* son buses por los cuales los nodos intercambian mensajes. Se identifican mediante un nombre único, y se emplean para comunicaciones anónimas de tipo publicación/subscripción. De esta manera se consigue desacoplar la creación de los mensajes de su consumo, ya que los nodos desconocen el extremo al que están conectados. Esto también hace posible múltiples publicadores y subscriptores a un cierto topic.

Los topics están íntimamente condicionados por el tipo de mensaje que transportan, siendo necesario que el tipo de mensaje del publicador coincida con el del subscriptor. Comprobar esta condición es una tarea llevada a cabo por el subscriptor, que se encarga de ejecutar el algoritmo MD5 sobre los mensajes intercambiados para verificar su integridad.

Como protocolo de transporte de los mensajes, ROS soporta TCP/IP (TCPROS) y UDP (UDPROS). Los nodos pueden negociar qué protocolo utilizar en tiempo de ejecución en función si se desea una mayor fiabilidad, en cuyo caso emplearán TCPROS, o bien una menor latencia, haciendo uso de UDPROS. En la Figura 2.7 se muestra una topología con varios publicadores y subscriptores a un mismo nodo:

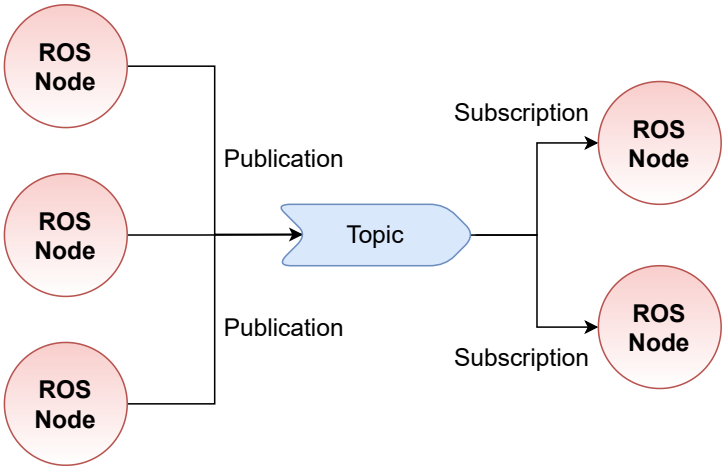

Figura 2.7: Topología con múltiples publicadores y subscriptores al mismo topic

2.2.1.3. Master

El maestro o ROS Master es el programa que se encarga de la gestión de la red. Proporciona los servicios para el registro de nodos y el intercambio de mensajes, manteniendo un seguimiento de los topics y servicios disponibles. Su papel principal es indicar a cada nodo la localización del resto de nodos, para que puedan conectarse entre sí y comunicarse de forma directa extremo a extremo. A continuación se muestra un breve ejemplo de la interacción entre el Master y dos nodos. En primer lugar el nodo *Camera* notifica al Master que quiere publicar en el topic *Images*:

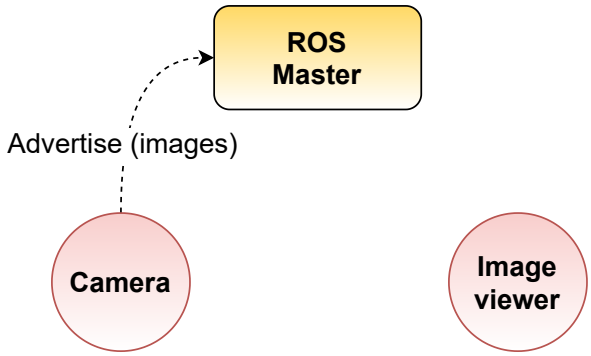

Figura 2.8: Publicación de un nodo en un topic

Una vez la petición es gestionada por el Master, *Camera* publica fotografías en el topic *Images*. Sin embargo, no existe ningún nodo suscrito a ese topic, por lo que no existe intercambio de mensajes. El nodo *Image viewer* solicita ahora al Master suscribirse a ese topic:

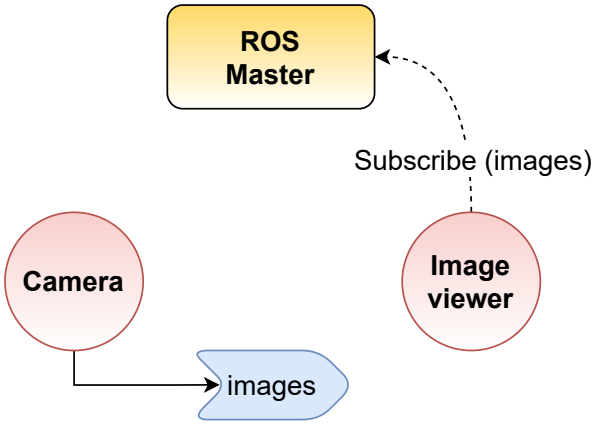

Figura 2.9: Subscripción de un nodo a un topic

Finalmente, el Master notifica a los nodos de la existencia de un publicador y suscriptor al mismo topic, con lo que comienza la comunicación entre extremos:

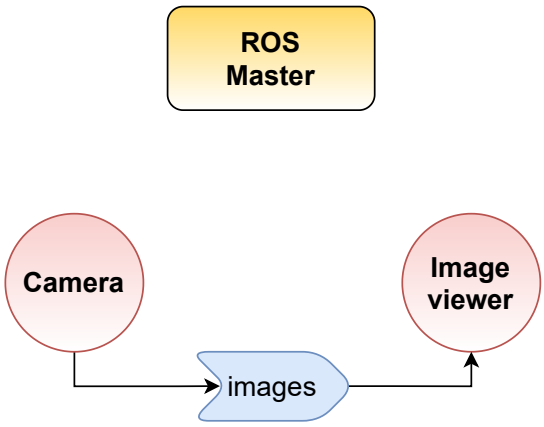

Figura 2.10: Intercambio de mensajes entre nodos

2.2.1.4. Servicios

La comunicación de tipo publicador-subscriptor presenta múltiples ventajas, al ser rápida, eficiente en términos de computación y muy flexible. Sin embargo, no es un método apropiado para esquemas clásicos de tipo cliente-servidor con llamada a procedimiento remoto (RPC), donde el cliente envía una petición a un servicio remoto y se queda esperando hasta recibir la respuesta. Para dar soporte a este tipo de comunicación, ROS incorpora los recursos necesarios para el despliegue de servicios. Los servicios en ROS tienen dos características principales:

- Son **bidireccionales**. Un nodo (cliente) envía información en forma de petición y se queda esperando a la respuesta. En el otro extremo, el servidor recibe los datos de la

petición del cliente, realiza con ellos las acciones o cómputos requeridos y devuelve la respuesta al cliente. Por tanto, existe un flujo de información en ambos sentidos, a diferencia de lo sucedido con el esquema publicador-subscriptor, donde no existe garantía de que el mensaje se haya recibido o ni siquiera de que exista un nodo subscriptor.

- Comunicación **individual** extremo a extremo. Cada petición es generada por un único cliente y dirigida un único servidor. Del mismo modo, la respuesta únicamente es recibida por el nodo que lanzó la petición.

Un servicio requiere de una definición de la estructura de datos que van a intercambiar los extremos, indicando los tipos de los campos de la petición y la respuesta en un fichero *.srv*. Los servicios, al igual que los topics, son registrados en el Master por el nodo que provee el servicio.

La Figura 2.11 muestra un esquema de comunicación mediante llamadas a servicios en ROS:

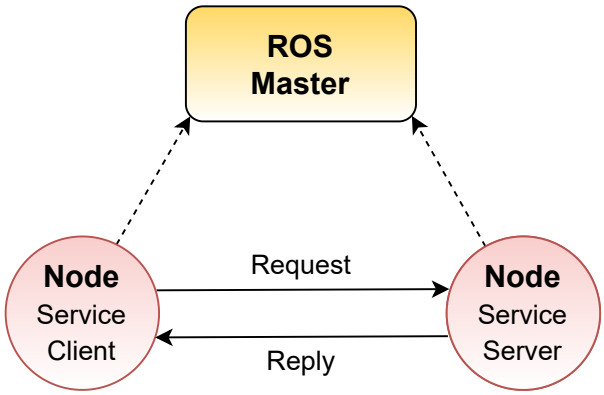

Figura 2.11: Comunicación mediante llamadas a servicios en ROS

2.3. Simulaciones multinivel

Como se ha mencionado anteriormente, hoy en día la simulación es una etapa fundamental dentro del proceso de diseño de cualquier sistema. A priori, cabría esperar que estas simulaciones fueran siempre lo más precisas posibles para emular la realidad lo máximo posible. La precisión de la simulación viene dada por grado de detalle provisto en el modelo virtual del sistema empleado, y por tanto surge una relación inversamente proporcional entre precisión y velocidad de simulación. Por ello, en ocasiones resulta necesario disponer de un modelo menos preciso pero que pueda simularse de manera más rápida, sobre todo

en las primeras etapas del proceso de diseño, donde el enfoque está más fijado en la funcionalidad general del sistema que en detalles más concretos. De esta manera, es posible dividir el problema abordándolo desde diferentes niveles de abstracción.

Por ejemplo, supongamos una flota de drones encargada de transportar mercancía entre dos fábricas cercanas. En una primera aproximación, puede ser conveniente emplear un modelo de dron simple que se mueva con velocidades constantes y simule de manera rápida, y poner un mayor esfuerzo en el desarrollo del código que envía las órdenes de control y seguimiento a los drones. Una vez esa parte esté implementada, se puede sustituir el dron por un modelo más preciso, que tenga en cuenta factores como el peso de la carga, velocidad del viento, fuerza de los motores, etc... para verificar que el código desarrollado sigue siendo válido, y de no ser así realizar los ajustes necesarios.

Sin embargo, no siempre resulta adecuado analizar los niveles de abstracción de manera separada. En el ejemplo anterior, es posible que exista un cuello de botella en el envío de la mercancía causado por alguno de los drones en particular, que han de realizar labores más cruciales que otros. En ese caso es interesante emplear modelos precisos para los elementos críticos y así analizar con mayor detalle esos momentos claves de la simulación.

Esta combinación de diferentes niveles de abstracción dentro de la misma simulación se denomina simulación multinivel. En este sentido, múltiples aproximaciones y herramientas han sido propuestas, como la co-simulación de diferentes modelos en simuladores separados [14] o el cambio dinámico de niveles de abstracción [15]. En un nivel inferior, también se aplican las técnicas de simulación multinivel en el ámbito de desarrollo hardware en FPGAs y DSPs [16].

En este trabajo, se adaptará la infraestructura VIPPE-ROS que se describirá más adelante para poder no sólo simular y estimar componentes robóticos, sino también ser capaces de realizar simulaciones multinivel de sistemas y servicios heterogéneos, de manera unificada sobre el mismo entorno de trabajo y de forma sencilla y rápida para el desarrollador.

CAPÍTULO

3 Integración de ROS en VIPPE

Una vez presentados los actores principales sobre los que se centra este trabajo, en este capítulo se explicarán los detalles estructurales y técnicos de las modificaciones realizadas en VIPPE para poder soportar código robótico descrito en ROS.

En un primer lugar, se valoró la opción de que todo ROS (el máster y los nodos) corriera sobre VIPPE como un ejecutable más. Sin embargo, para ello es necesario que tanto ROS como todas las librerías de las que depende sean compilados con VIPPE. Esto no sería problema en el caso de ROS al estar sus fuentes disponibles de manera abierta, pero en cambio sí que resulta inviable para sus dependencias, ya que en numerosas ocasiones los códigos fuente de éstas no son accesibles. Por tanto, se descartó esta opción y se optó por dejar ambos elementos independientes, haciendo uso de las facilidades de ROS para realizar simulaciones.

Pese a estar los dos dominios separados, el objetivo de este trabajo es que conformen un todo. Cada componente ROS formará parte de la simulación como un elemento más, y el hecho de ser ROS no debe de ser impedimento para que VIPPE pueda tener un dominio total sobre el mismo. Para ello, en primer lugar es necesario que VIPPE controle el avance del tiempo de ROS, de manera que los distintos componentes puedan interactuar entre sí correctamente independientemente de su naturaleza. Así mismo, las funciones de la librería de ROS deben de poder ser empleadas con normalidad, tal y como sucedería en la realidad. De estos dos aspectos fundamentales derivan una serie de consideraciones que han de tenerse en cuenta y serán abordadas en éste capítulo. En muchas de ellas se ha optado, como solución, por generar clases de envoltura o *wrappers* de las funciones ROS específicas

que causan alguno de estos efectos. Estos wrappers se encargan de mantener bajo control la ejecución de dichas funciones, realizando diversas tareas en función de su cometido. Finalmente, al tratarse de una plataforma de análisis de prestaciones, será necesario estimar los tiempos de estas funciones de librería de ROS para poder anotarlos y tenerlos en cuenta en el cómputo general de la simulación, aprovechando los wrappers generados para el caso anterior.

Una vez descritos de manera general los aspectos más importantes, a continuación se explicarán en detalle los mecanismos implementados para la simulación de componentes ROS en VIPPE.

3.1. Control temporal

Uno de los aspectos fundamentales a abordar es la sincronización entre VIPPE y ROS. Al tratarse de dos dominios de simulación diferenciados, es necesario comprender cómo funcionan los tiempos en ambos dominios antes de entrar en detalles acerca del mecanismo de sincronización implementado.

El kernel de VIPPE permite avanzar a los hilos que se ejecutan bajo su control hasta un máximo de tiempo para asegurarse que no se produce un desajuste temporal entre ellos. Este máximo temporal se denominas *slice*. Por defecto, VIPPE tiene definido un slice de 10 milisegundos. Al comienzo de cada slice, VIPPE planifica las tareas y las manda ejecutar durante el tiempo que requieran. Si una vez transcurrido el tiempo del siguiente slice las tareas no han terminado de ejecutar, se realiza una nueva replanificación y se lanzan de nuevo, repitiendo éste proceso hasta que todas las tareas hayan terminado. Entre cada slice de tiempo, VIPPE sincroniza los hilos y acumula las métricas necesarias para el reporte final en término de ciclos e instrucciones ejecutadas, energía, tiempo empleado...

En la Figura 3.1 se muestra un esquema del avance temporal en VIPPE:

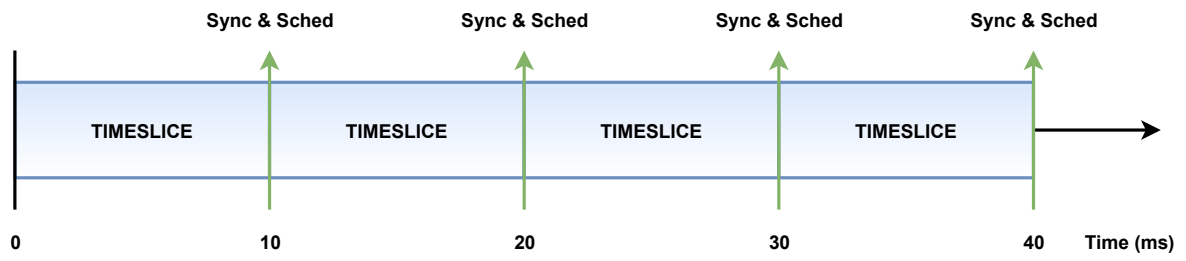

Figura 3.1: Avance de tiempo en VIPPE

En el caso de ROS, por defecto se emplea el tiempo de *host* para la ejecución de los nodos y la gestión de sus aspectos temporales, como frecuencias, esperas, etc.. Esto es, el tiempo real o *wall-clock time*. Por tanto, con este tiempo únicamente se podrían realizar ejecuciones en tiempo real a una velocidad de 1 segundo por segundo. Para solucionar este problema y dar cobertura a simulaciones más rápidas, ROS soporta el empleo de un tiempo simulado como base temporal. Esto se consigue estableciendo el parámetro de simulación */use_sim_time* a verdadero, y publicando el tiempo simulado en un topic ofrecido por el Master denominado */clock*. Este topic acepta mensajes de tipo *rosgraph_msgs::Clock*, formados por un campo de segundos y otro de nanosegundos.

Una vez descrito el funcionamiento temporal de ambos dominios de simulación, resulta necesario establecer un mecanismo de sincronización entre ambos. Para ello, se ha desarrollado un nodo ROS llamado *Sync Clock*. Este programa establece una comunicación bidireccional entre ROS y VIPPE por medio de sendas colas de mensajes POSIX. Realiza dos funciones principales:

1. Cada vez que VIPPE incrementa su tiempo en un slice temporal, manda a *Sync. Clock* el tiempo simulado en ese momento (*actual_sim_time*), el cual lo procesa adaptándolo a mensajes de tipo *rosgraph_msgs::Clock* y lo publica en el topic */clock*.

2. Por cada iteración anterior, este nodo envía de vuelta a VIPPE el tiempo actual de simulación de ROS (*ros::Time::now()*), de manera que VIPPE se bloquea hasta que dicho tiempo coincide con el último tiempo que envió al *Sync. Clock*. Esto es debido a que ROS tarda una pequeña fracción de tiempo en actualizar su tiempo simulado tras haberlo recibido en el topic */clock*, y de no bloquearse, VIPPE seguiría avanzando su tiempo generando una desincronización entre ambos. Este caso se produce cuando la carga ejecutada por VIPPE es baja y por tanto su avance de tiempo es más rápido de lo que tarda ROS en actualizar su tiempo.

El esquema de sincronización descrito se muestra en la Figura 3.2 :

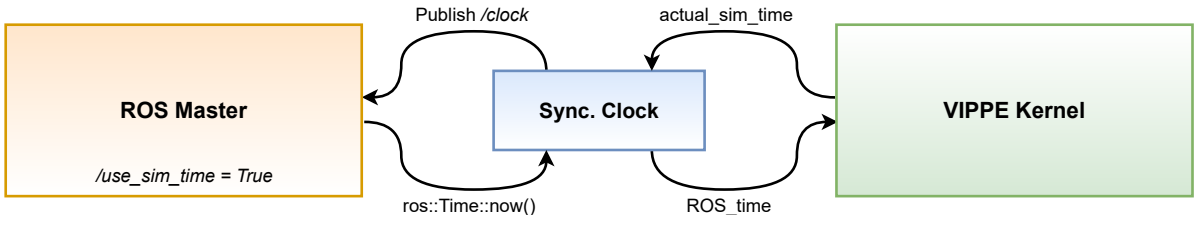

Figura 3.2: Sincronización base entre VIPPE y ROS

Otro aspecto relevante que requiere de una sincronización especial entre ROS y VIPPE sucede al inicializar un nodo ROS. Para inicializar un nodo, se llama a la función *ros::init* y se crea un manejador de nodo o *ros::NodeHandle*. Durante una fracción de tiempo desde la inicialización, el nodo aún no ha recibido el tiempo actual de simulación de ROS y tiene un tiempo local igual a cero. El tiempo cero en ROS es un valor inválido que ha de tratarse de manera diferente, dado que al no producirse avance de tiempo, los nodos están bloqueados durante esa fracción temporal. Para VIPPE por tanto son tareas inactivas, y el núcleo continúa su avance generando una desincronización entre los nodos ROS simulados y el resto de componentes del sistema, si los hubiera. Para solventar este problema, es necesario bloquear el avance de VIPPE durante una fracción de tiempo hasta que el nodo haya recibido un valor de tiempo correcto. Por ello se ha creado una clase *wrapper* de *ros::NodeHandle* para bloquear VIPPE, esperar hasta contar con un valor de tiempo válido y finalmente volver a desbloquear VIPPE. Para comprobar la validez temporal del nodo se hace uso de la función *ros::Time::isValid()*, la cual devuelve verdadero en el momento en el que el tiempo recibido es distinto de cero y por tanto válido. El mecanismo de sincronización descrito se muestra en la Figura 3.3:

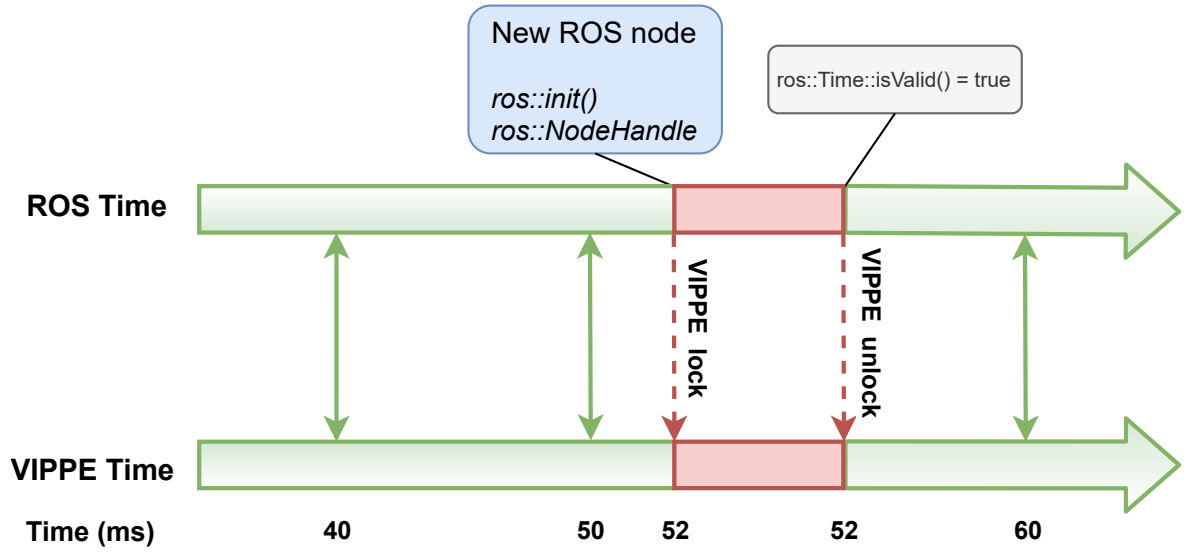

Figura 3.3: Sincronización VIPPE-ROS ante la creación de un nodo

Mediante los dos mecanismos de sincronización descritos anteriormente, se asegura un ajuste dinámico de tiempo entre ROS y VIPPE, en función de la carga simulada.

3.2. Desacoplo de funciones

Una de las ventajas del empleo de una infraestructura como ROS, aparte del despliegue y gestión de las comunicaciones entre nodos, es el poder hacer uso de las funciones propias de librería que incorpora. Estas funciones se llaman desde los nodos pero se ejecutan fuera del núcleo de simulación de VIPPE, y por tanto deben ser tratadas de forma específica. Es un caso similar a las llamadas a sistema operativo en VIPPE, donde se hace uso del sistema operativo del *host* para emular al sistema operativo del *target*. En concreto, requieren especial atención las funciones bloqueantes que paralizan al nodo que las llama, como son las esperas (*ros::Time::sleep()* o *ros::Duration::sleep()*) o las llamadas a servicio. Las dos primeras realizan una espera del periodo o duración especificado, mientras que la última se queda esperando hasta que se recibe la respuesta. La función de refresco de las publicaciones (*ros::spinOnce()*) realiza internamente una llamada a la cola de publicaciones, por lo que también ha de considerarse como llamada bloqueante.

De llamarse de forma directa, todas estas funciones provocarían un bloqueo en la simulación. Esto es debido a que el nodo realiza la llamada a la función y se queda esperando. Al pararse, se detiene también el avance de tiempo en VIPPE, y por tanto también en ROS. En conclusión, la función llamada no llega a ejecutarse ya que ROS no está incrementando su tiempo. Es por tanto, un bloqueo doble entre VIPPE y ROS. Para solventar este problema, la solución propuesta consiste en desacoplar del núcleo de simulación de VIPPE a las tareas ROS que realizan llamadas bloqueantes a funciones externas, permitir que ejecuten de manera independiente, y volver a engancharlas cuando retornen. Así mismo, mientras están desenganchadas, VIPPE continúa ejecutando el resto de tareas que tenga planificadas de manera normal. La función de desacoplo creada se llama *VIPPE_detach* y la de enganche *VIPPE_attach*.

Por todo ello, se han implementado clases de envoltura o *wrappers* de las clases ROS que implementan las funciones bloqueantes, sustituyendo el prefijo de namespace "*ros::*"por "*UC_ros_*". De esta manera, es posible realizar las operaciones de enganche y desenganche antes y después de la llamada a la función. Para que este cambio sea un proceso transparente para el desarrollador, se ha modificado el compilador de VIPPE para que en el comienzo de proceso de compilación se realice un reemplazo automático de las clases originales por las clases de envoltura. Para indicar que la fuente compilada se trata de un código ROS, y por tanto realizar el reemplazo, se ha de añadir el argumento - *-vippe-ROS* en la línea de compilación. A modo de ejemplo, en el Código 3.1 se muestra el código correspondiente al wrapper de la clase *ros::Duration*:

```
class UC_ros_Duration: public ros::Duration {
public:
  UC_ros_Duration(double time):ros::Duration(time){timeUs=(int)(time*1e6);};
  ~UC_ros_Duration(){};
  bool sleep(){
        VIPPE_detach(((long long int)timeUs)*1000);
        bool ret=ros::Duration::sleep();
        VIPPE_attach();
        return ret;
  }
private:
  int timeUs;
};
```

Código 3.1: Wrapper de la clase *ros::Duration*

Para ilustrar con mayor claridad el funcionamiento de estas clases, en la Figura 3.4 se muestra un ejemplo de un nodo ROS que realiza operaciones en un bucle a una frecuencia de 1Hz. En el caso original, mostrado en la parte superior, al realizar la llamada a la función *sleep* para esperar el periodo correspondiente, se produce el bloqueo doble anteriormente descrito, bloqueando por tanto la simulación. En la parte inferior, se muestra el esquema propuesto, donde la clase original *ros::Rate* ha sido reemplazada de manera automática por la clase propia *UC_ros_Rate*. En la llamada a la función *sleep*, se realiza en primer lugar el desacoplo, se llama a la función original, esperando hasta que retorne para después enganchar el proceso a VIPPE y continuar con la simulación:

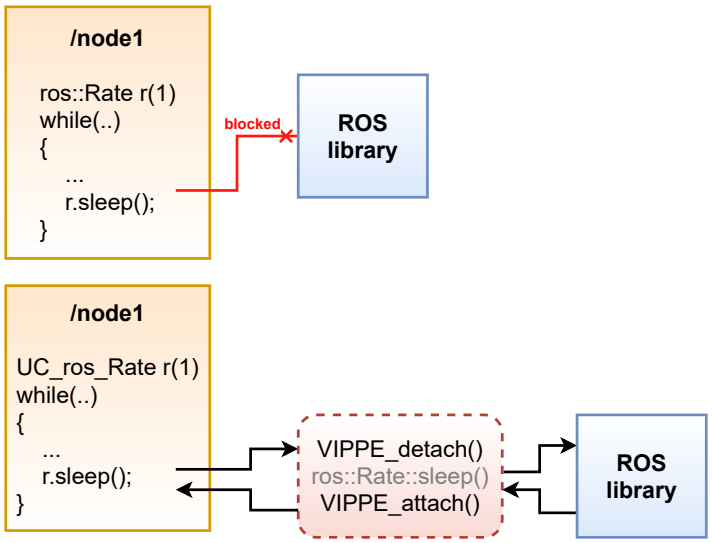

Figura 3.4: Ejemplo del empleo de las funciones de enganche y desenganche

Un efecto colateral que surge al desacoplar procesos de VIPPE, es que el núcleo de simulación sigue aumentando su tiempo y potencialmente lo hace de manera más rápida al ser la carga a ejecutar menor. Por tanto puede suceder que en funciones como el *sleep* descrito anteriormente, al retornar la función y volver a engancharse al núcleo, éste haya ejecutado más tiempo del que debería, y por tanto se produzca una desincronización entre los nodos ROS y el kernel de VIPPE. Por ejemplo, supongamos que en el tiempo 1000 ms de simulación se quiere hacer un *sleep* de 10 milisegundos. En principio, al finalizar el *sleep* la simulación debería estar en el tiempo 1010 ms. Sin embargo, al desenganchar la tarea, VIPPE ejecuta de manera más rápida, y al retornar se encuentra con que el tiempo de simulación es 1050 ms.

Para solucionar este problema, en la función de desenganche se pasa como argumento el tiempo máximo hasta el que puede ejecutar VIPPE. En cada vuelta del kernel, se comprueba si existen tareas desenganchadas y si el tiempo de simulación ha alcanzado el tiempo máximo a ejecutar, y de ser así, se bloquea el núcleo de VIPPE. Cuando se ejecute la función de enganche de la tarea correspondiente, se desbloquea VIPPE y continúa la simulación. Esto puede observarse en el Código 3.1, donde la función *VIPPE_detach* recibe como argumento el tiempo de sleep en nanosegundos, es decir, el tiempo máximo hasta el que puede ejecutar VIPPE antes de bloquearse.

3.3. Anotación

Otro aspecto abordado en este trabajo es la estimación temporal de las funciones más empleadas de ROS. Dado que son funciones de librería y se ejecutan desacopladas de VIPPE, no es posible realizar una anotación en el propio código en base al análisis del mismo. Por tanto, si se quieren proporcionar métricas de estimación de rendimiento precisas, es necesario incluir sus tiempos de ejecución en alguna parte del código.

Para ello se hace uso de las funciones de envoltura descritas anteriormente. Al ser llamadas, estas funciones realizarán una anotación en base a una carga predefinida. La estimación de estas cargas en base a diferentes parámetros será el objeto de esta sección.

Para obtener los resultados se mide el tiempo empleado por cada función en una muestra suficientemente grande de veces. Este tiempo viene determinado por dos factores principales: la frecuencia del procesador y la arquitectura del mismo. Para cubrir un rango suficientemente grande de posibilidades, se obtienen los tiempos ejecutando las funciones en un PC (procesador Intel i5 con arquitectura x86_64) y en una placa RaspBerry Pi 4 (procesador ARM Cortex-A72 con arquitectura ARMv8). Con estas dos arquitecturas es

posible cubrir gran parte de los potenciales escenarios, ya que los nodos si se ejecutan en PC este va a ser probablemente un Intel, y en caso de correr sobre un embebido el procesador dominante es el ARM. Para el caso de las frecuencias, se obtienen los tiempos con dos frecuencias diferentes para cada tipo de procesador:

- Intel (x86_64): 1600MHz y 3000MHz
- ARM (ARMv8): 600MHz y 1500MHz

De esta manera es posible realizar una extrapolación a cualquier frecuencia deseada con un error menor que suponiendo una relación puramente lineal entre tiempo de ejecución y frecuencia del procesador.

Para realizar la extrapolación en frecuencias se emplea el método que se describe a continuación.

1. Los resultados obtenidos se expresa de forma matemática de acuerdo a expresiones lineales del tipo:
$$Time(ns) = A \cdot x + B.$$
2. Una vez obtenidos A y B, se descomponen a su vez otros dos términos dependientes de la frecuencia, de forma que:
$$A = a \cdot f(MHz) + b$$
$$B = c \cdot f(MHz) + d$$
3. De esta forma, la expresión final resultante viene dada por:
$$Time(ns) = (a \cdot f(MHz) + b) \cdot x + (c \cdot f(MHz) + d)$$

De entre todas las funciones de la librería de ROS, se han analizado aquellas que son más relevantes. Las llamadas a servicio no han sido consideradas en este apartado ya que el tiempo de ejecución de la llamada es, en general, despreciable respecto al tiempo de ejecución del propio servicio. Las funciones estimadas se dividen en dos grupos. Por un lado están aquellas funciones cuyo tiempo de ejecución tiene alguna dependencia con factores externos (en la ecuación anterior, dependientes de x), y por otro lado las funciones que no tienen este tipo de dependencias, es decir, valores constantes ($x = 0$). Las primeras contarán con términos en a,b,c,d y x, mientras que las segundas únicamente dependerán de c y d, siendo x=c=d=0.

A continuación se explican las funciones estimadas y el procedimiento empleado.

3.3.1. Funciones dependientes

Este grupo de funciones requieren de un tiempo de uso de CPU que es dependiente de un factor variable, y por tanto han de modelarse de acuerdo a ecuaciones matemáticas que reflejen esas dependencias. Las funciones analizadas en este apartado son *publish, sleep* y el proceso *rosmaster*.

3.3.1.1. Publish

La función para el envío de mensajes desde un nodo publicador hasta los nodos subscriptores es *ros::Publisher::publish*. Tal y como se describió en el apartado 2.2.1.3, la publicación se realiza punto a punto entre el publicador y cada uno de los suscriptores. Por tanto, el tiempo empleado por esta función será directamente proporcional al número de suscriptores de un determinado topic.

Para realizar la estimación de la función, se ha ejecutado iterativamente un número muy elevado de veces, variando el número de nodos suscritos al topic publicado, y midiendo el tiempo empleado por el procesador haciendo uso de los relojes de POSIX.

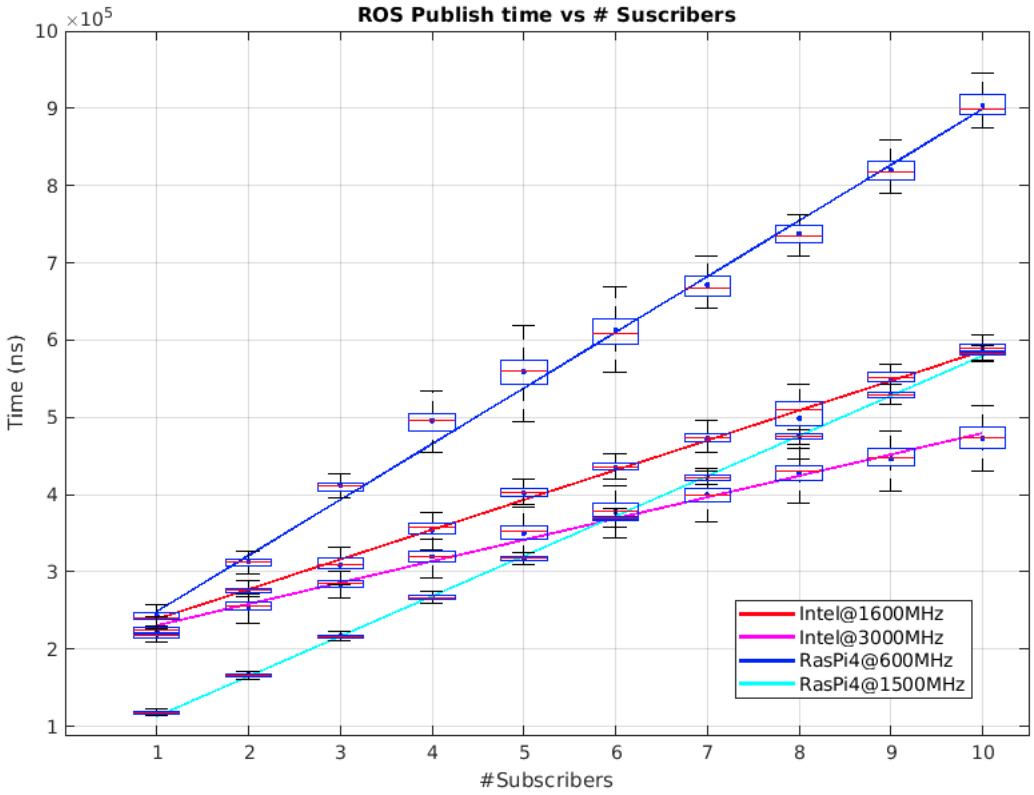

Figura 3.5: Estimación de la función *ros::Publisher::publish* dependiendo del número de suscriptores

Los resultados obtenidos se reflejan en la Figura 3.5, donde se muestra el tiempo empleado por publicación en nanosegundos en función del número de suscriptores. Se observa como en ambas arquitecturas el tiempo empleado es muy similar en el caso de un nodo suscriptor, y que en ambos casos la tendencia resulta lineal. Sin embargo, para el caso del ARM, el tiempo crece de manera más rápida que en el caso del Intel.

De estos resultados se derivan las siguientes expresiones realizando un ajuste lineal, las cuales servirán para modelar la carga de la función, siendo S el número de suscriptores:

$$t_{intel@1600MHz} = 38565{,}54 \cdot S + 200852{,}35 \ \ ns \tag{3.1}$$

$$t_{intel@3000MHz} = 27667{,}69 \cdot S + 203595{,}65 \ \ ns \tag{3.2}$$

$$t_{ARM@600MHz} = 72256{,}46 \cdot S + 177082{,}84 \ \ ns \tag{3.3}$$

$$t_{ARM@1500MHz} = 51874{,}62 \cdot S + 61464{,}55 \ \ ns \tag{3.4}$$

Finalmente, de las ecuaciones anteriores se obtienen las expresiones finales para cada arquitectura en función de la frecuencia del procesador en MHz:

$$t_{intel}(ns) = (-7{,}78 \cdot f(MHz) + 51020) \cdot S + (1{,}959 \cdot f(MHz) + 197717) \tag{3.5}$$

$$t_{ARM}(ns) = (-22{,}656 \cdot f(MHz) + 85844) \cdot nSubs + (-128{,}46 \cdot f(MHz) + 254162) \tag{3.6}$$

Adicionalmente se realizaron pruebas variando la tasa de publicación, el tamaño del mensaje publicado y el tamaño de los búferes de envío. Sin embargo, no se detectó que ninguno de estos factores influyera significativamente en los resultados obtenidos.

3.3.1.2. Sleep

Las funciones de espera en ROS se llaman desde dos clases principales, *ros::Rate* y *ros::Duration*:

- La clase *Rate* implementa una función de espera en función de la tasa indicada de su correspondiente periodo. Se emplea habitualmente para que el nodo que la instancia realice operaciones a una cierta frecuencia, para por ejemplo, enviar mensajes con una tasa determinada.
- La clase *Duration* implementa una función de espera en función de una duración en segundos indicada en el constructor. Esta función se usa generalmente para realizar esperas de una duración determinada, como la función de *sleep* de POSIX, y habitualmente sin ser llamada de forma reiterada, como por ejemplo para bloquear un nodo durante un cierto tiempo conocido hasta que otra parte del sistema haya arrancado.

Estas funciones operan por *polling*, chequeando periódicamente si el tiempo del sistema ha alcanzado el valor deseado para reanudar la tarea. Por tanto, existe una dependencia directa entre el tiempo de espera y la carga que genera en el procesador: a mayor valor de la espera más número de comprobaciones han de realizarse y por tanto más uso de CPU.

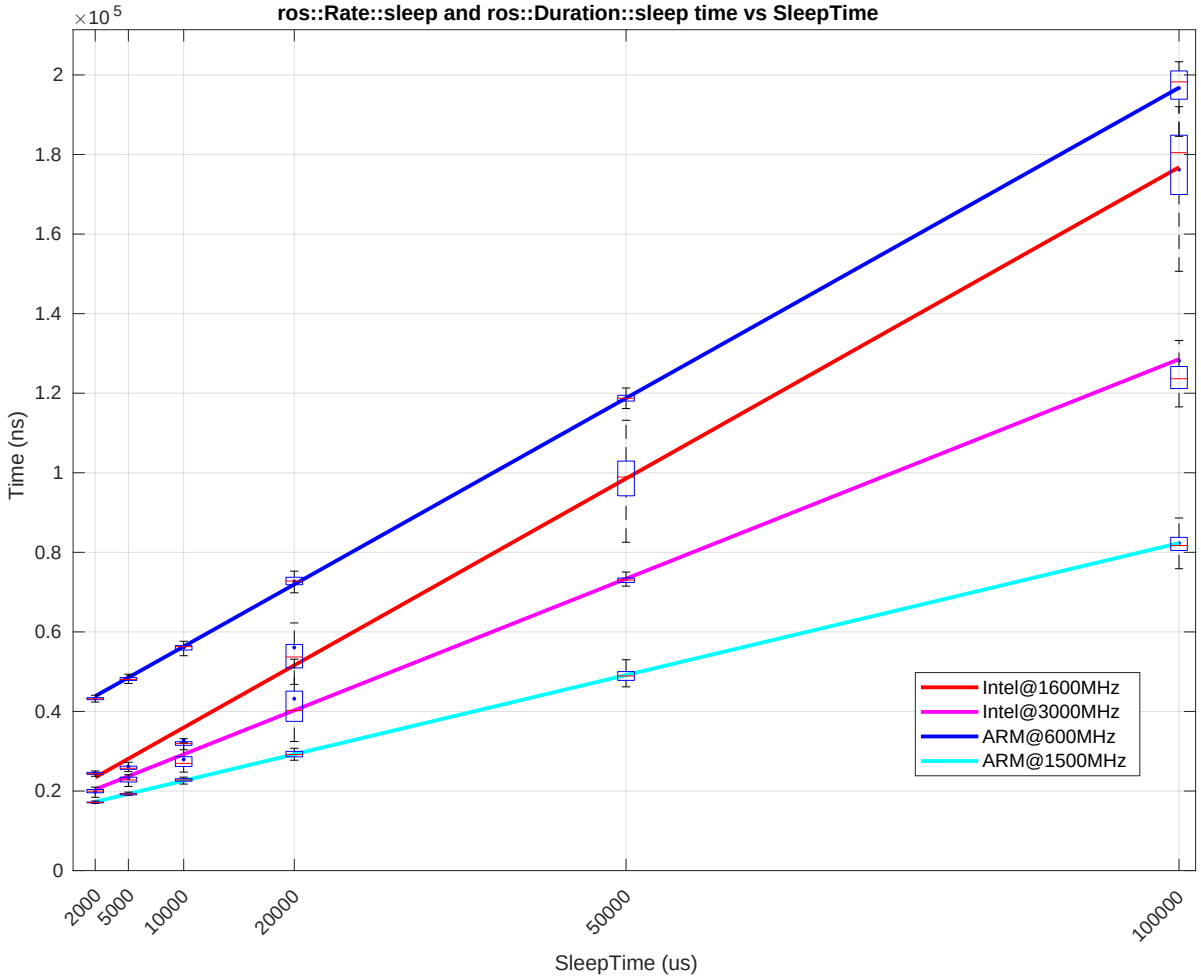

Figura 3.6: Estimación temporal de las funciones *ros::Rate::sleep* y *ros::Duration::sleep* dependiendo del tiempo de espera

La Figura 3.6 muestra la relación entre el tiempo empleado por el procesador y el tiempo de espera, ajustándose al comportamiento de una función lineal de un término en la forma $ax + b$. A continuación se indican las expresiones obtenidas, siendo T el periodo en microsegundos:

$$t_{intel@1600MHz} = 1{,}5650 \cdot T + 20300 \;\; ns \tag{3.7}$$

$$t_{intel@3000MHz} = 1{,}5606 \cdot T + 40725 \;\; ns \tag{3.8}$$

$$t_{ARM@600MHz} = 1{,}1033 \cdot T + 18200 \ \ ns \tag{3.9}$$

$$t_{ARM@1500MHz} = 0{,}6635 \cdot T + 15965 \ \ ns \tag{3.10}$$

De forma idéntica al apartado previo, se obtienen las expresiones finales resultantes de las ecuaciones anteriores para cada arquitectura:

$$t_{intel} = (-0{,}33 \cdot f(MHz) + 2{,}0925) \cdot T + (-1{,}498 \cdot f(MHz) + 22697) \ \ ns \tag{3.11}$$

$$t_{ARM} = (-0{,}996 \cdot f(MHz) + 2{,}1587) \cdot T + (-27{,}51 \cdot f(MHz) + 57231) \ \ ns \tag{3.12}$$

3.3.2. Funciones sin dependencias

En este grupo de funciones se incluyen principalmente aquellas de inicialización y registro del nodo, que son llamadas habitualmente una única vez al comienzo de la ejecución del nodo. También se analizan las funciones de refresco de las colas de mensajes y obtención del tiempo de sistema. En el caso de este grupo de funciones, al realizar el análisis de los tiempos no se ha detectado ninguna dependencia relevante con factores externos. Por tanto, la mejor estimación posible consiste en realizar un promedio de los resultados obtenidos dentro de una muestra lo suficientemente grande de ejecuciones.

3.3.2.1. Funciones de inicialización y registro

En la Tabla 3.2 se muestran los tiempos medios de ejecución de estas funciones obtenidos para las dos arquitecturas a las frecuencias anteriormente mencionadas:

	Time (ns)			
Function	**Intel@1600MHz**	**Intel@3000MHz**	**ARM@600MHz**	**ARM@1500MHz**
init	433923	300054	18795260	16195670
NodeHandle	4137052	2322393	5583100	4885054
serviceClient	58706	47143	78275	68099
advertise	316862	195434	591090	525178
suscribe	1078574	731497	1445835	1360615
advertiseServer	142290	121073	189720	24403

Tabla 3.1: Estimación temporal para las funciones ROS de inicialización y registro

A partir de estos datos, se obtienen los términos c y d resolviendo los sistemas de ecuaciones:

	Intel		ARM	
Function	**c**	**d**	**c**	**d**
init	-95,62	586916	-1856,85	21766220
NodeHandle	-1296,19	6210948	-498,60	6380867
serviceClient	-8,26	71921	-7,27	89905
advertise	-86,73	455637	-47,08	666420
suscribe	-247,91	1475233	-60,87	1543229
advertiseServer	-15,16	166538	-17,62	217907

Tabla 3.2: Factorización en función de la frecuencia para las funciones ROS de inicialización y registro

3.3.2.2. Spin

Los mensajes que se mandan a través de un topic al que un nodo está suscrito se reciben a través de una función de retrollamada o *callback*. El nodo que desea suscribirse pasa como argumento en la suscripción la función de callback a la que desea que le lleguen los mensajes. Cuando llega un mensaje, ROS no llama directamente a la función, sino que encola la petición hasta que se realiza una llamada a *ros::spinOnce*. Esto permite, entre otros, recibir los mensajes con una cierta periodicidad únicamente cuando el nodo lo desee, por ejemplo para no sobrepasar su capacidad de procesamiento.

Pese a no depender de factores externos, se ha observado que el tiempo empleado por esta función es mayor la primera vez que es llamada por un nodo que el resto de las veces. Los resultados promedios de tiempo obtenidos se muestran en la Tabla 3.3:

Processor	**Time (ns)** **First execution**	**Tiempo (ns)** **Rest of executions**
Intel @ 1600MHz	21500	6377
Intel @ 3000MHz	20380	6270
ARM @ 600 MHz	54381	21860
ARM @ 1500 MHz	24403	9048

Tabla 3.3: Estimación temporal para la función *ros::spinOnce*

De manera idéntica a los casos anteriores, se obtienen los términos *c* y *d* resolviendo los sistemas de ecuaciones:

	Intel		ARM	
Function	**c**	**d**	**c**	**d**
spinOnce (First)	-0,80	22780	-21,41	88642
spinOnce (Rest)	-0,08	6499	-9,15	36502

Tabla 3.4: Factorización en función de la frecuencia para la función *ros::spinOnce*

3.3.2.3. ros::Time::now()

Dentro de la ejecución de un nodo ROS, es habitual necesitar conocer el tiempo actual del sistema, para por ejemplo obtener diferencias temporales de cara a realizar una espera o simplemente depurar el código. La función de la librería ROS que permite conocer ese tiempo se denomina *ros::Time::now()*. A continuación se muestran los tiempos de ejecución obtenidos para esta función en las plataformas y frecuencias estudiadas:

Processor	**Time (ns)**
Intel @ 1600MHz	1476
Intel @ 3000MHz	1069
ARM @ 600 MHz	4385
ARM @ 1500 MHz	2060

Tabla 3.5: Estimación temporal para la función *ros::Time::now()*

Del mismo modo que en los casos anteriores, tras realizar la factorización, se obtienen los términos *c* y *d* para realizar la anotación

	INTEL		ARM	
Function	**a**	**b**	**a**	**b**
ros::Time::now()	-0,29	1941	-1,66	7042

A modo de resumen de esta sección y la anterior, la Tabla 3.6 recoge las funciones ROS consideradas en este trabajo sobre las cuales se han creado wrappers para evitar bloqueos y/o realizar anotaciones, indicando para cada caso si se tratan de funciones bloqueantes y si han sido anotadas según el procedimiento descrito:

Function	Blocking	Annotated
ros::Rate::sleep	Yes	Yes
ros::Duration::sleep	Yes	Yes
ros::Publisher::publish	No	Yes
ros::ServiceClient::call	Yes	No
ros::NodeHandle	No	Yes
ros::NodeHandle::advertise	No	Yes
ros::NodeHandle::advertiseService	No	Yes
ros::NodeHandle::subscribe	No	Yes
ros::NodeHandle::serviceClient	No	Yes
ros::init	Yes	Yes
ros::spinOnce	No	Yes

Tabla 3.6: Funciones ROS consideradas en este trabajo

3.4. Simulación en tiempo real

Para finalizar este capítulo se presenta la última modificación realizada en VIPPE y de gran utilidad a la hora de simular sistemas robóticos. Aunque en la inmensa mayoría de las veces se busca que las simulaciones sean lo más rápidas posible, en multitud de ocasiones es conveniente y hasta imprescindible que las simulaciones de estos sistemas se produzcan en tiempo real (RT), es decir, que el tiempo simulado (TSo) sea igual al tiempo de simulación (TSn) de manera que la velocidad de simulación sea de 1 segundo por segundo (TSo/TSn=1). Esta situación se da, entre otros, cuando es necesaria la visualización del robot en el espacio mediante un simulador 3D o cuando se requiere interacción humana con el sistema, por ejemplo para el control manual de un robot. Este último caso se denomina *Human-In-The-Loop*, y modifica el resultado de la simulación en función de las acciones del usuario.

Para soportar la simulación de sistemas en tiempo real, se ha modificado el kernel de VIPPE para alinear el avance del tiempo simulado y de simulación. Por un lado se guarda el tiempo del sistema en el arranque de la simulación y se le acumula el tiempo actual de simulación. Por otro lado, se obtiene el tiempo actual del sistema en cada vuelta del núcleo, y se realiza una espera de la diferencia para que ambos coincidan y avancen de manera coordinada. Este mecanismo se activa añadiendo la opción - *-vippe-realtime* al lanzar la simulación en VIPPE.

Para que la simulación en tiempo real sea posible, ha de cumplirse que el tiempo de cómputo de las tareas planificadas durante un slice temporal sea menor que el valor del

propio slice. De lo contrario, la simulación será más lenta que 1 segundo por segundo. El modelo matemático de esta condición se representa en la siguiente figura:

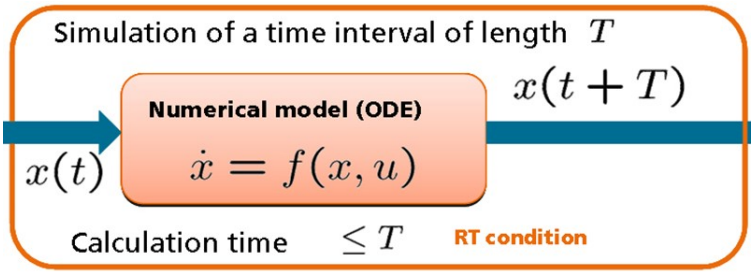

Figura 3.7: Condición de tiempo real [17]

CAPÍTULO

4 Modelos robóticos: Particularización para drones

Dentro de los sistemas robóticos, el abanico de posibles ejemplos es realmente amplio, y cada uno de ellos cuentan con sus consideraciones particulares. Abordar todos ellos es una tarea excesiva, y debido a ello y al proyecto dentro del cual se enmarca, en este trabajo se ha decidido particularizar para un tipo concreto de robots: los drones.

Un dron es un sistema robótico volador no tripulado, que puede ser controlado mediante radiofrecuencia de manera total, parcialmente controlado y parcialmente autónomo, o totalmente autónomo. Son sistemas que han sufrido una enorme expansión en los últimos años en ámbitos tan diversos como el militar, comercial o recreativo, ya que permiten realizar operaciones muy variadas sin necesidad de una gran infraestructura, a un bajo coste, y a veces evitando poner en riesgo vidas humanas.

Como se ha descrito en capítulos anteriores, la simulación mediante modelos virtuales se presenta altamente conveniente en este tipo de sistemas previo a su despliegue final. Para poder simular sistemas de este tipo, es necesario contar con dos elementos claramente diferenciados:

- El autopiloto: es el software que se ejecuta sobre el dron, y recibe los comandos desde el piloto o la estación de tierra y la información de los sensores para enviar la potencia requerida a cada uno de los rotores del dron en base a la operación requerida en cada momento.

- El modelo físico del dron: se encarga de asociar las señales enviadas por el autopiloto a los motores con su velocidad y el movimiento del dron en el aire, y realimentar el

lazo cerrado generando las señales correspondientes para los sensores. La precisión del modelo y las ecuaciones físicas que emplea determinan el grado de realismo del vuelo del dron.

Dentro de la simulación global, el dron será un componente más dentro de un conjunto de elementos que interactúan entre sí. Por ejemplo, un dron que se comunica con una estación de tierra que le envía órdenes y monitoriza su estado. Por ello, en sistemas de estas características formados por componentes de diversa naturaleza, resulta conveniente dividirlos en dos tipos diferentes. Por un lado están los componentes que se desean desarrollar y/o optimizar durante el proceso de diseño, y por otro lado hay elementos que no se modifican y se toman como componentes de librería, previamente diseñados por el propio desarrollador o por entidades externas.

Para el caso particular de los drones, al diseñar un sistema que hace uso de ellos no parece muy lógico ponerse a desarrollar un autopiloto completo para el dron. Por tanto, para el diseño final, se empleará uno comercial ya creado, permitiendo centrar el esfuerzo en el diseño del resto de componentes. Sin embargo, organizando el flujo de trabajo desde el comienzo del desarrollo, sí que resulta relevante contar con diferentes modelos de dron, de más simple a más complejo. De esta manera, al principio del desarrollo se emplearán los modelos más simples para obtener simulaciones rápidas y poder definir las necesidades generales del resto de componentes de manera ágil, y a medida que avanza el diseño emplear los modelos más complejos para realizar el perfilado final. En cada etapa se busca detectar el mayor número de errores de diseño, para poder realizar las modificaciones necesarias lo antes posible. Es por tanto un buen ejemplo de la utilidad de la simulación multinivel.

4.1. Niveles de simulación de componentes genéricos

En este trabajo se proponen cuatro niveles de abstracción, reflejados en la Tabla 4.1 y que se explican a continuación.

El nivel más alto de simulación (MN) deberá permitir al diseñador realizar simulaciones rápidas con un bajo esfuerzo en el diseño. La funcionalidad de los componentes será mínima y vendrá descrita a grandes rasgos, estableciendo la estructura sobre la cual se basará el resultado final. En este caso, en general no es relevante obtener métricas de tiempo, energía o prestaciones precisas, sino que se buscará que cumpla con la funcionalidad básica deseada, simulando de la manera más rápida posible. En todo caso, si se quieren extraer unos resultados temporales preliminares, puede ser interesante añadir cargas predefinidas para obtener una primera visión general del sistema (MC).

Level	Code	Time/Energy
MN	minimal	no
MC	minimal	constant
FC	full	constant
FD	full	data dependent

Tabla 4.1: Niveles de abstracción para los componentes C++ y rospp

Una vez esta primera etapa ha sido superada, el código puede ser desarrollado de manera más profunda y detallada. Para el caso de sistemas en tiempo real, como drones, resulta fundamental considerar aspectos temporales y de energía. Sin embargo, esta no es una tarea sencilla y rápida. Por ello, en este punto, se proponen dos niveles: por un lado combinar el código con las cargas constantes del nivel anterior (FC), y por otro realizar el análisis dinámico y completo del código empleando la anotación de VIPPE (FD).

Adicionalmente, en sistemas complejos de este tipo, resulta conveniente que cada componente pueda simular en niveles de abstracción diferentes como los descritos en la Tabla 4.1. Esto da lugar a simulaciones heterogéneas, donde por ejemplo los componentes que se están desarrollando requieren simular con un nivel alto de detalle (FD) mientras que los de librería puede ser suficiente con que simulen en el nivel FC.

4.2. Niveles de simulación del dron

Como se ha introducido anteriormente, en el caso del autopiloto del dron resulta conveniente contar con dos códigos diferentes. En el modelo más sencillo y funcional, el dron se mueve a velocidades constantes y realiza el movimiento de cada trayectoria de un único paso. Por otro lado, el modelo de dron completo genera todas las señales de los motores necesarias para el simulador físico, calculando todas las posiciones intermedias, ángulos, orientación y consumo de energía. Para este trabajo el autopiloto escogido es ArduCopter, un autopiloto ampliamente utilizado tanto a nivel de usuario como en investigación, al ser de código abierto y licencia GLPv3.

El hecho de introducir ROS en la simulación tiene ciertas implicaciones que es necesario considerar. ROS está pensado para emplear principalmente comunicación de tipo publicador-suscriptor, donde los suscriptores requieren que el flujo de información se produzca de manera constante y periódica por los publicadores. Por tanto, el primer modelo de dron propuesto, que realiza los movimientos completos en un único salto, no resulta adecuado para ser empleado con ROS. Por ello, como parte de este trabajo se ha diseñado un

modelo de dron en ROS que publica con un flujo de datos constante los datos aproximados relativos a la posición del dron. Los niveles de abstracción del dron aparecen en la Tabla 4.2.

Level	Drone model
F	Functional
A	ArduCopter

Tabla 4.2: Niveles de abstracción del modelo de dron

4.3. Entorno de simulación multinivel

Una vez detallados los niveles de simulación, en este apartado se describe el entorno de simulación diseñado. Como se ha comentado, el autopiloto del dron que se empleará en el servicio final es ArduCopter. Para las comunicaciones, ArduCopter hace uso de MAVLink, un protocolo de comunicación para el control del dron y envío de telemetría. MAVLink puede ser utilizado en comunicaciones radio (por ejemplo desde un mando radiocontrol o una estación de tierra), o bien mediante puertos serie si se comunican dos placas físicas (por ejemplo, una placa embarcada en el dron que ejecute un programa de tratamiento de imagen). Es el protocolo más utilizado para comunicación en el ámbito de drones, ya que es ligero y soporta el intercambio de mensajes de múltiples tipos. Por tanto, es la interfaz que emplea el dron para la comunicación con el exterior. Sin embargo, su uso directo no es trivial para un desarrollador software y requeriría un conocimiento profundo del protocolo. Por todo ello, se decide introducir MAVROS en el entorno de simulación.

MAVROS [18] es un nodo ROS que actúa como driver de comunicaciones para autopilotos que hacen uso de MAVLink. Ofrece una traducción de mensajes MAVLink a una serie de topics y servicios en ROS, por lo que la comunicación entre el dron y el exterior se realiza de manera mucho más sencilla para el desarrollador. Esta topología permite mover la interfaz de comunicaciones del dron de MAVLink a ROS, y tratar al dron en su conjunto como un nodo ROS. Este esquema se muestra en la Figura 4.1:

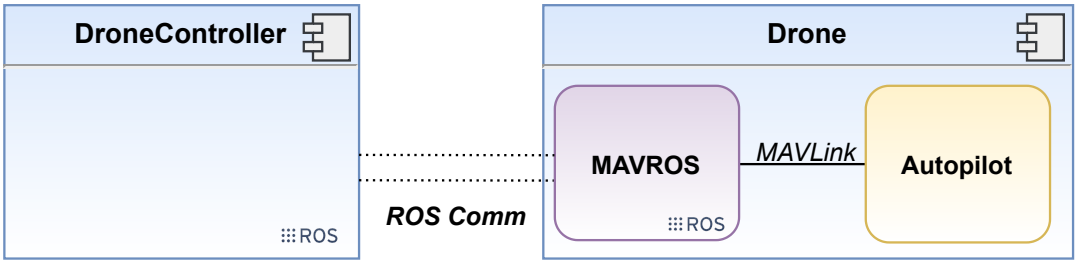

Figura 4.1: Ejemplo de comunicación con ArduCopter

La simulación física del dron se realiza con el simulador SITL (Software in the Loop). Este simulador emula de manera realista la aerodinámica del dron para obtener una simulación de gran precisión.

Una vez que todos los elementos del marco de trabajo han sido introducidos, la topología general del entorno de simulación se muestra en la Figura 4.2:

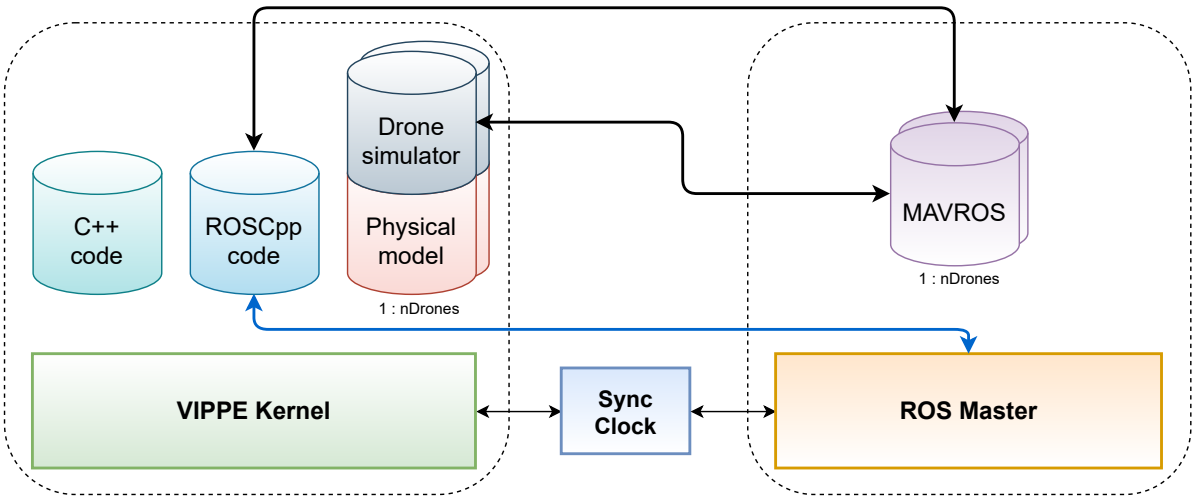

Figura 4.2: Entorno de simulación multinivel para drones

En la figura se muestran de manera separada los dos dominios de simulación. Por un lado, la funcionalidad del sistema, llevada a cabo por los componentes C++, rospp y el autopiloto con su simulador físico, se ejecutan bajo el control de VIPPE. Por otro lado, el nodo MAVROS se ejecuta por separado controlado únicamente por el máster de ROS. En la figura se refleja la interacción entre el código rospp y el máster, del cual obtiene la base temporal para su ejecución. Adicionalmente, el entorno de simulación multinivel ha de ser capaz de realizar simulaciones con múltiples drones, ya que cada vez más servicios requieren de una flota o enjambre de drones volando al mismo tiempo, muchas veces interactuando entre ellos. Por tanto, se indica esta multiplicidad en los componentes que se replicarán en función del número de drones necesarios: el simulador de drones con su modelo físico, y el nodo MAVROS, dado que cada dron requiere de un MAVROS propio.

La idea que subyace a este entorno es proporcionar una gran flexibilidad a la hora de realizar simulaciones, pudiendo el desarrollador escoger aquel escenario que necesite en cada momento. Por tanto, en función del nivel de abstracción a simular, algun componente de los indicados podrá estar o no presente en la simulación. Por ejemplo, en el caso de la simulación con un dron funcional, el autopiloto y el simulador físico se reemplazarán por el modelo simple de dron en ROS, y el nodo MAVROS será omitido, comunicando

directamente el código roscpp (*DroneController* en la Figura 4.1) con el modelo de dron. En el caso de la simulación completa con ArduCopter, éste será el modelo de autopiloto y todos los componentes mostrados formarán parte de la simulación.

CAPÍTULO

5 Ejemplo de aplicación: Servicios basados en drones

Una vez realizada la adaptación de VIPPE para soportar código robótico ROS como infraestructura base, y definidos los diferentes niveles de simulación de drones, se ha diseñado un caso de uso para poder verificar el correcto funcionamiento del sistema completo y obtener métricas relevantes dentro del proceso de diseño del servicio. Este ejemplo se ejecutará sobre el entorno de trabajo diseñado de cara a poder realizar simulaciones multinivel.

5.1. Caso de uso

El caso de uso desarrollado para este trabajo consiste en un servicio de reparto de paquetería mediante una flota de drones. Es un ejemplo claro de los beneficios de la simulación durante el proceso de diseño de este tipo de servicios, ya que de probarse directamente sobre los drones reales sin haberse verificado su correcto funcionamiento es realmente arriesgado y pondría en peligro la integridad de los propios drones y de la carga que transportan.

El caso de uso propuesto cuenta con 4 elementos fundamentales para realizar la funcionalidad deseada del servicio:

- Central de envíos: Se encarga de procesar la petición de compra del cliente, enviar su dirección a un generador de rutas, y enviar al controlador del dron la ruta a seguir hasta el destino.
- Generador de rutas: Recibe la ubicación actual del dron y el destino, y genera una ruta con una serie de *waypoints* para ser enviada al dron.

- Controlador del dron: Es un nodo ROS encargado de enviar las órdenes necesarias al dron y monitorizar su vuelo para comprobar que ha realizado la entrega de manera satisfactoria. Para añadir una carga real en la simulación, este componente realiza un cómputo del algoritmo de detección de bordes *sobel* sobre una imagen VGA recibida cada segundo desde el dron para emular un sistema de detección de obstáculos.

- Dron: Es el dispositivo mecatrónico encargado de transportar la mercancía en función de las órdenes recibidas. Cuenta con un nodo ROS que recibe las órdenes del controlador y envía información acerca del estado del dron: posición GPS, altura, batería...

El esquema general del servicio descrito se muestra a continuación:

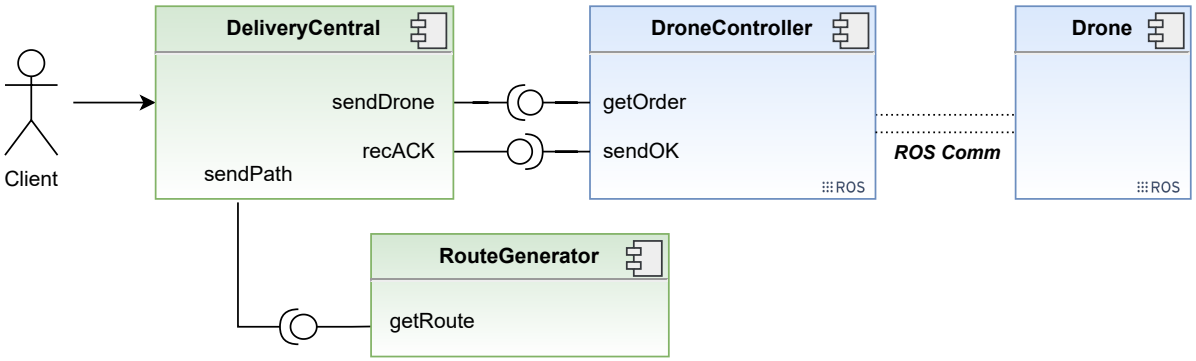

Figura 5.1: Esquema del servicio

La comunicación entre los componentes del sistema se realiza empleando dos métodos:

- Si ambos componentes son nodos ROS, se emplea la infraestructura de ROS para las comunicaciones. En el servicio propuesto en la Figura 5.1 sería el caso de las comunicaciones entre el controlador y el dron, denotadas como *ROS Comm*.

- Si no, la comunicación entre componentes se realiza mediante llamadas a procedimiento remoto a través de interfaces provistas y requeridas. Es el caso de la comunicación entre la central, el generador de rutas y el controlador.

Cuando se hace uso de ROS, para evitar conflictos entre nodos de la misma naturaleza que tienen el mismo nombre (por ejemplo, múltiples instancias del mismo dron), se hace uso de los *namespaces* de ROS. Por tanto, para este sistema, los nodos y topics se organizan como */uav1, /uav2,* ..., en función del dron con el que están relacionados.

5.2. Resultados experimentales

Una vez presentado el caso de uso y el entorno de trabajo sobre el que será ejecutado, para demostrar el correcto funcionamiento de la infraestructura VIPPE-ROS, los modelos de dron y la plataforma multinivel, en esta sección se procede a detallar los resultados obtenidos realizando diversas simulaciones. En primer lugar, es conveniente conocer la precisión de la estimación del código ROS descrita en la Sección 3.3, por lo que se realizará un estudio de la misma comparando los valores temporales estimados con los valores reales medidos en el propio código. Seguidamente, se realizarán múltiples simulaciones del caso de uso para demostrar el tipo de resultados que es posible obtener a partir de la plataforma de simulación multinivel.

5.2.1. Precisión de la estimación

Para verificar la precisión de las anotaciones descritas en la Sección 3.3 se simula el caso de uso completo empleando Arducopter como autopiloto. En este caso, el interés reside en la estimación de rendimiento del código ROS, es decir, el controlador del dron (*DroneController*). Por tanto, para que no exista interferencia en los resultados obtenidos, el resto del código C++ se simula sin haber sido anotado empleando las técnicas de simulación nativa, de manera que las llamadas a función de ROS es la única carga tenida en cuenta. Por otro lado, las medidas se realizan ejecutando el código real sobre la propia plataforma y midiendo y acumulando el tiempo de uso real de procesador de las llamadas ROS empleando el reloj de proceso de POSIX. Los resultados obtenidos para las plataformas y frecuencias estudiadas se muestran en la Tabla 5.1:

		Time (ns)		
	Frequency	**Estimated**	**Measured**	**Error (%)**
Intel	1600	485,873,399	582,581,163	16.60
	3000	318,626,359	461,462,380	30.95
ARM	600	521,466,358	904,953,908	42.38
	1500	233,965,058	398,265,784	41.25

Tabla 5.1: Comparación entre tiempo estimado y tiempo medido de ROS

Tal y como se esperaba, existe un error relevante en el tiempo estimado de las funciones ROS. Este error está causado por una falta de precisión en la estimación, errores en las medidas temporales realizadas en el propio código, o a una combinación de ambas. Se

observa que el error cometido en el caso de la plataforma Intel es menor, pero más variable con la frecuencia, mientras que en el caso del ARM presenta un error mayor pero con menos variación.

A pesar de ello, al ser el error menor del 100 % en todos los escenarios, el error total en un ejemplo completo y anotado se reduce incluyendo estas anotaciones, ya que si no se tienen en cuenta (es decir, el tiempo de ejecución de ROS se considera despreciable) el error cometido sería en ese caso del 100 %. Este impacto depende por tanto de la cantidad de código ROS que incluya la aplicación. En un extremo, si el porcentaje de código ROS (%rc) respecto al total de la aplicación es del 0 %, el error de la estimación (tee) sería el propio de la simulación nativa (nse), típicamente en el rango del 10-25 %. En el otro extremo, si el código ROS supone el 100 % del total, el error cometido (ree) sería el mostrado en la tabla superior, en un rango de entre el 15-45 %. Generalizando, el error cometido estaría entre los extremos de la siguiente expresión:

$$tee = \%rc \cdot ree + (1 - \%rc) \cdot nse \tag{5.1}$$

Con todo ello, teniendo en cuenta el global de la aplicación empleada como caso de uso en este trabajo, se obtiene la Tabla 5.1:

	Frequency	**Total application (ms)**	**ROS code (%)**	**Impact of ROS estimation error (%)**	**Impact of no estimation**	**Improvement**
Intel	1600	22,875	2.55	0.42	2.55	83.5 %
	3000	22,018	2.10	0.65	2.10	69.0 %
ARM	600	213,446	0.42	0.18	0.42	57.1 %
	1500	87,688	0.45	0.19	0.45	57.8 %

Tabla 5.2: Impacto del error de ROS en las estimaciones

Como se observa, el impacto del error en la estimación del tiempo de ejecución de las funciones ROS es pequeño, y en todo caso mucho menor que considerándolo despreciable. Con todo ello, empleando las anotaciones descritas en este trabajo se obtiene una mejora del error de hasta el 83 % para los casos donde el porcentaje de código ROS es mayor.

5.2.2. Simulaciones multinivel

La flexibilidad de la plataforma permite un gran número de combinaciones a simular, y todas ellas tienen sentido en algún contexto específico dentro del proceso de diseño. En esta

sección se pretende emular un flujo de diseño convencional en el cual se van añadiendo detalles en el modelo de manera progresiva, partiendo de un modelo mínimo hasta llegar al resultado final.

En primer lugar, se estudia la dependencia entre la velocidad de simulación y el número de CPUs disponibles en el PC donde se ejecute la plataforma de simulación, siendo en todos los casos el número de drones simulados 8. Es relevante proporcionar resultados en función del número de CPUs ya que sirve para comprobar si la plataforma de simulación puede ser ejecutada en cualquier tipo de PC, sin que su potencia de cómputo sea un factor determinante. Los resultados obtenidos se muestran en la Figura 5.2.

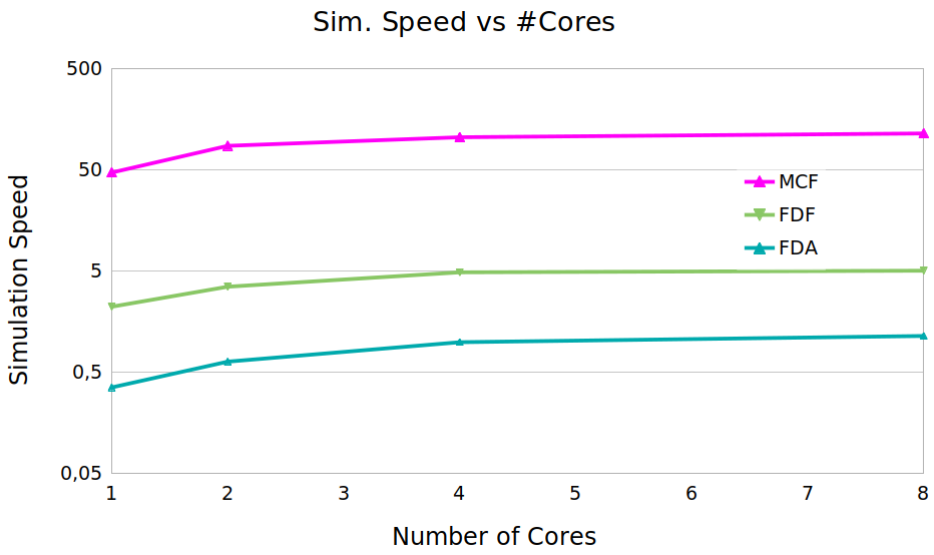

Figura 5.2: Relación entre número de cores y velocidad de simulación

Siguiendo con la nomenclatura empleada en la Tabla 4.1 y Tabla 4.2, la curva superior se corresponde con un nivel MCF, empleando un dron funcional y una versión del código C++ mínima (sin sobel) con unas cargas predefinidas. En el flujo de diseño, el objetivo de este nivel es verificar de manera preliminar que el código de aplicación es correcto. Como se observa, la velocidad de simulación es muy elevada, por encima de 100 s/s a medida que se aumentan los cores empleados.

Tras haber verificado el funcionamiento del sistema, se desarrolla el código real de los componentes y estos pueden ser anotados empleando las técnicas de simulación nativa. Se corresponde por tanto con el nivel FDF, en el cual se sigue manteniendo el modelo funcional de dron pero ya con la versión completa del código anotado. Se observa en este caso como al

introducir código real la velocidad de simulación cae un orden de magnitud, estableciéndose entre valores de 2 y 7 segundos por segundo empleando 1 y 8 cores, respectivamente. Finalmente, se reemplaza el modelo de dron funcional por el autopiloto de ArduCopter, obteniendo la curva FDA. Se observa como la velocidad de simulación vuelve a caer de manera considerable, siendo únicamente superior a 1 segundo por segundo cuando se emplean 4 o más cores.

En el experimento anterior, la velocidad de simulación cayó de manera radical cuando los 8 drones empleados se modelaron con ArduCopter. En el siguiente experimento, se analiza el impacto del modelo de dron en la velocidad de simulación. Para ello, se establece un escenario con 8 drones funcionales en el cual estos van siendo sustituidos de manera progresiva por drones reales, es decir, modelados con ArduCopter. Además, este experimento demuestra como la plataforma permite mezclar diferentes modelos con diferente nivel de detalle para realizar simulaciones heterogéneas. El modelo de los componentes es FC (es decir, se emplea el código completo empleando cargas de tiempo y energía constantes). Los resultados se muestran en la Figura 5.3:

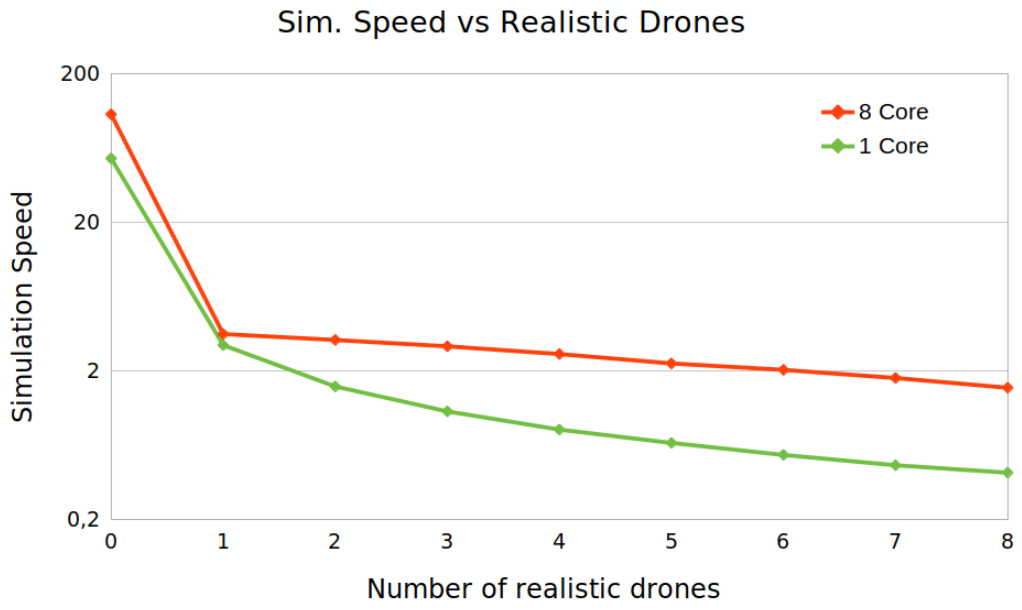

Figura 5.3: Relación entre el número de drones realistas y velocidad de simulación

Cuando todos los drones son funcionales el nivel simulado es FCF. En este caso, la velocidad de simulación está comprendida entre valores de 50 y 100 en función del número de CPUs empleadas. En el momento en el que se reemplaza uno de los drones funcionales por un modelo real, la velocidad de simulación cae bruscamente un orden de magnitud. A medida que aumenta el número de drones reales, la velocidad sigue decreciendo, pero de manera menos acusada que en el primer caso. Este impacto es, tal y como se espera, mayor cuantos menos cores estén disponibles para ejecutar la simulación.

Para recalcar la dependencia entre la velocidad y el número de drones reales en la simulación, en la Figura 5.4 se muestra, en función del número de CPU, cómo ésta se ve reducida a medida que se aumentan los drones a simular.

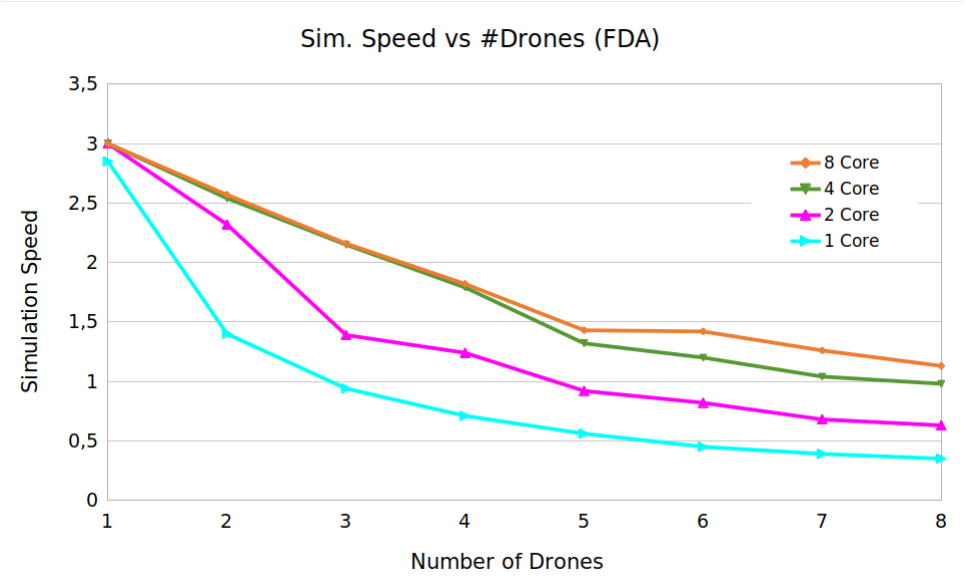

Figura 5.4: Relación entre número de drones y velocidad de simulación en función del número de cores

Tal y como se explicó en la Sección 3.4, el entorno de trabajo diseñado soporta simulaciones en tiempo real con una velocidad de 1 segundo por segundo. Para demostrar el correcto funcionamiento de esta característica, se realizan diversas simulaciones en un nivel FDA (código completo anotado con ArduCopter) aumentando el número de drones en la simulación. En la Figura 5.5 se muestra la relación en tiempo de uso real de CPU en función del número de drones. Se observa claramente cómo el tiempo de CPU aumenta de manera mucho más rápida a medida que el número de cores disponibles es menor:

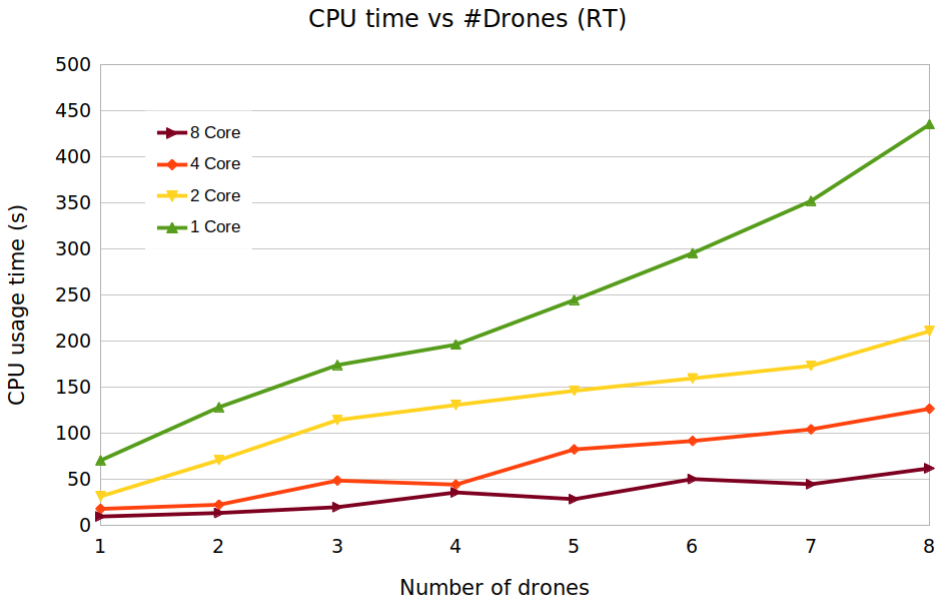

Figura 5.5: Tiempo de CPU en función del número de drones en la simulación en tiempo real

Directamente relacionada con la figura anterior, en la Figura 5.6 se muestra el porcentaje de uso de CPU para el mismo experimento:

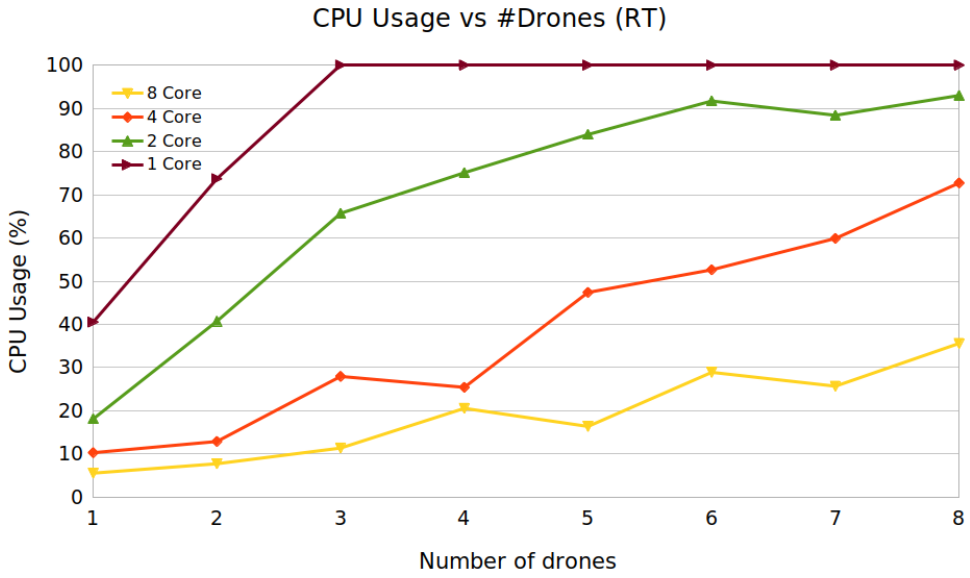

Figura 5.6: Uso de CPU en función del número de drones en simulación en tiempo real

Se observa que empleando un único core el uso de CPU sube rápidamente al 100 % a

partir de 3 drones, momento a partir del cual la velocidad de simulación baja por debajo de 1 segundo por segundo. En el otro extremo, empleando 8 cores es posible simular los 8 drones con apenas el 30 % de uso de CPU.

A modo de comparativa, en la Figura 5.7 se muestran los resultados del mismo experimento pero realizando una simulación normal, lo más rápido posible (*AFAP*). Se observa como el uso de CPU para 1 core ya parte de casi el 100 % simulando un único dron, demostrando el correcto funcionamiento de la simulación en tiempo real, ya que es consecuencia de bloquear el núcleo de simulación para adaptar su avance al tiempo real de sistema. Este mismo motivo causa que el resto de curvas también tengan valores superiores al caso en tiempo real. La forma en diente de sierra de algunas de estas curvas se debe a que, en ciertas combinaciones, al añadir elementos a la simulación la CPU es capaz de paralelizar ciertas tareas, reduciendo la carga total.

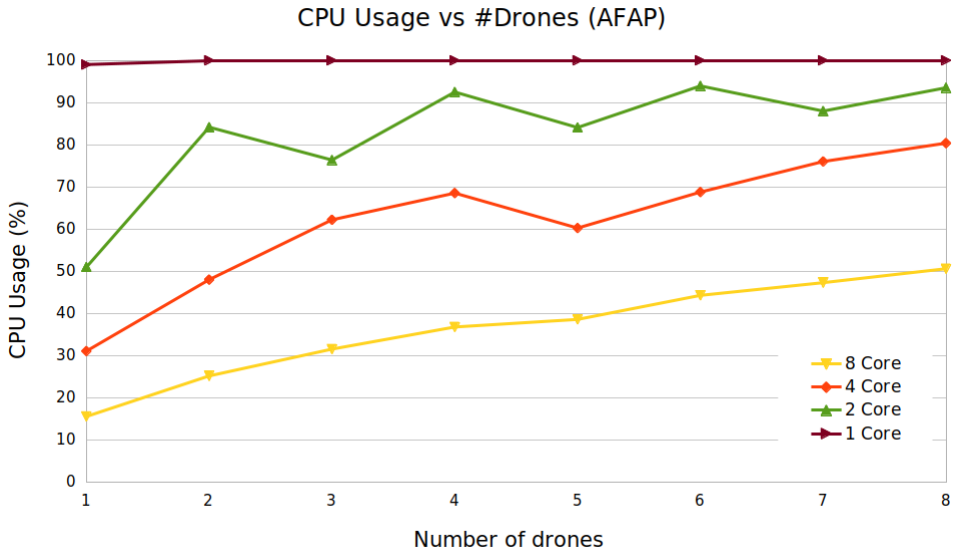

Figura 5.7: Uso de CPU en función del número de drones en simulación normal

Para finalizar esta sección, en la última gráfica se muestra a modo de resumen el impacto de la carga simulada (en este caso número de drones) frente a la velocidad de la simulación, y en función del nivel de abstracción simulado. Se observa como el nivel superior, con código mínimo y modelo de dron funcional obtiene unas velocidades de simulación superiores a 100 segundos por segundo en todos los casos. La curva intermedia presenta el mismo escenario pero añadiendo el código real del sobel en la simulación, reduciendo un orden de magnitud la velocidad de simulación. Finalmente, empleando un modelo de dron realista y anotando el código real para ser estimado, la velocidad de simulación vuelve a caer otro

orden de magnitud. Este ejemplo muestra de manera clara el proceso a seguir dentro de un proceso de diseño, partiendo de un modelo muy sencillo el cual es posible probar y mejorar de manera muy rápida, e ir añadiendo detalles y funcionalidades hasta obtener el diseño final.

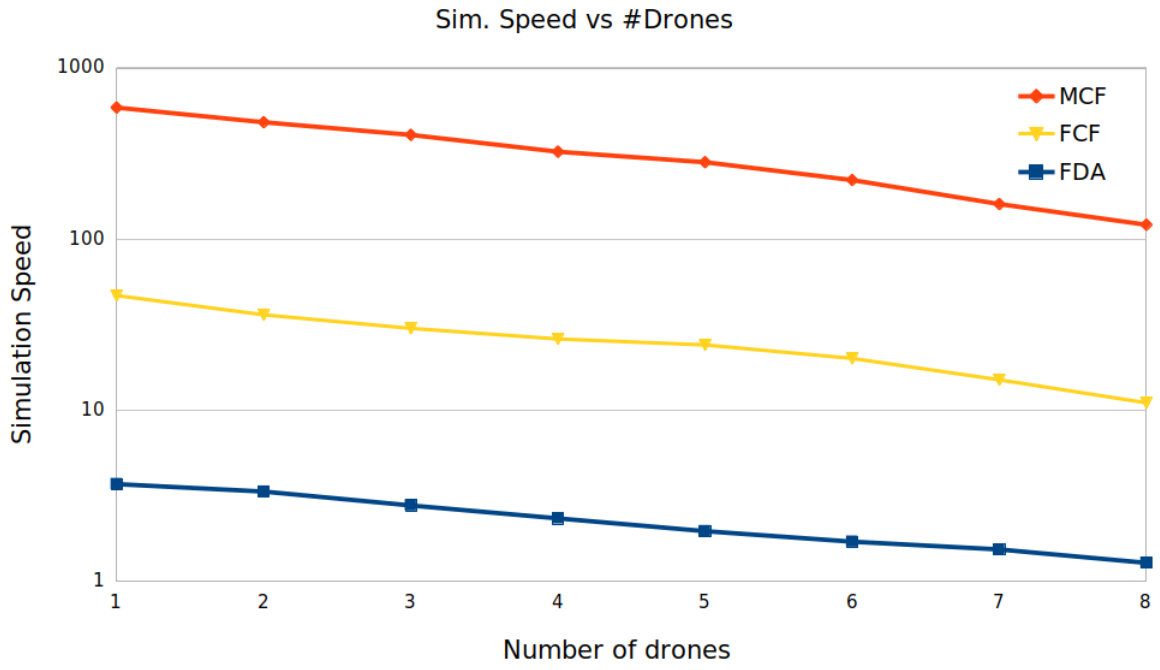

Figura 5.8: Relación entre número de drones y velocidad de simulación en función del nivel de abstracción

CAPÍTULO

6 Conclusiones

Este trabajo propone una plataforma para simular código robótico basado en ROS empleando el simulador VIPPE desarrollado por el Grupo de Ingeniería Microelectrónica. Para verificar su correcto funcionamiento, se ha diseñado un caso de uso de un servicio basado en drones con varios ejemplos de código a diferentes niveles de abstracción para poder realizar simulaciones heterogéneas multinivel.

En primer lugar se ha realizado un estudio acerca del estado del arte de las técnicas de simulación nativas, sobre las cuales se basa VIPPE, y de la infraestructura ROS y sus funcionalidades más importantes. Una vez conocidos los detalles de funcionamiento de ambos sistemas se diseñaron e implementaron los mecanismos de sincronización necesarios para poder simular código ROS en VIPPE. Esto incluye un proceso para sincronizar los relojes y diferentes procedimientos dependiendo de las funciones ROS ejecutadas. Además, se crearon clases de envoltura de las funciones ROS para poder desenganchar su ejecución del núcleo de VIPPE y evitar condiciones de bloqueo. Seguidamente, al tratarse de una plataforma que permite realizar un análisis de prestaciones, se realizó una estimación del tiempo empleado por las funciones ROS para poder realizar anotaciones en función del procesador y su frecuencia. Finalmente, se añadió la característica de poder realizar simulaciones en tiempo real para permitir la interacción humana con el sistema (*Human-in-The-Loop*).

Para demostrar que estas aportaciones funcionan de manera correcta, se diseña un caso de uso de un servicio de paquetería que emplea drones para el reparto de la mercancía. En dicho servicio el dron incorpora un nodo ROS que se comunica con un controlador que es a su vez otro nodo ROS. Se emplean dos modelos de dron diferentes, uno realista que emplea el autopiloto ArduCopter y otro funcional con un nivel mínimo de detalle que

permite simulaciones veloces. El resto son componentes estándar C++ que se comunican mediante mecanismos RPC. Se han establecido diferentes niveles de abstracción en función de la complejidad del código C++ y del modelo de dron empleado. Finalmente, mediante diversos experimentos, se ha recreado el flujo de diseño de un servicio de este tipo desde un prototipo sencillo hasta el sistema final, complejo y preciso, verificando el correcto funcionamiento del sistema.

6.1. Contribuciones

Los resultados obtenidos en este trabajo, enmarcado dentro del proyecto europeo ECSEL JU Comp4Drones, han sido empleados en los entregables dentro del mismo proyecto. Además, han derivado en una serie de contribuciones científicas que se muestran a continuación.

Congresos internacionales

[MGP1] J. Merino, R. Gomez, H. Posadas, E. Villar, *"Multilevel host-compiled simulation framework for ROS-based UAV services using ArduCopter"*, en 2021 Design of Circuits and Integrated Systems (DCIS). Pendiente de revisión.

[MGP2] J. Merino, R. Gomez, H. Posadas, E. Villar, *"Modeling and Performance Estimation of Robotic Systems using ROS: Application to drone-based Services"*, en 2021 Forum on Specification & Design Languages (FDL). Pendiente de revisión.

Seminarios internacionales

"Model-Driven simulation and performance analysis of ROS-based systems", en HiPEAC 2021. Charla invitada.

"Model-Driven Design of CPSoSs: Application to drone-based services", en 2021 Cyber-Physical Systems and Internet of Things (CPS&IoT'2021).

6.2. Líneas futuras

Aunque el modelo de simulación se crea de manera sencilla, este proceso debería realizarse de manera automática desde un modelo único. Por ello, dentro del proyecto, el grupo está desarrollando una herramienta para realizar esta generación partiendo de un modelo creado en UML en base a la metodología S3D [19]. De esta manera, será posible

realizar las simulaciones a diferentes niveles de abstracción de manera rápida y automática, permitiendo la configuración de diferentes aspectos relativos a las comunicaciones entre componentes (modelos de computación) y parámetros propios de los componentes.

Como línea de mejora, sería interesante perfeccionar las anotaciones estimadas en la Sección 3.3 para mejorar su precisión y cubrir un mayor número de funciones. Para ello, se podrían utilizar técnicas de Machine Learning e Inteligencia Artifical para aumentar la exactitud de las estimaciones, ya que estas tecnologías permiten tener en cuenta un mayor número de variables que las consideradas. Además, mediante estos métodos sería posible obtener datos relativos al consumo real de la ejecución de las funciones sobre los diferentes *targets* para poder también proporcionar métricas de consumo. Por tanto, sirva este trabajo para establecer las pautas sobre las que realizar estas tareas y así poder alcanzar una plataforma de simulación más precisa y robusta.

Bibliografía

[1] "Overview - ecsel comp4drones." https://www.comp4drones.eu/project-info/overview/.
(Accessed on 03/08/2021).

[2] "Copter home — copter documentation." https://ardupilot.org/copter/.
(Accessed on 03/08/2021).

[3] H. Posadas Cobo, *Estimación de prestaciones para exploración de diseño en sistemas embebidos complejos HW/SW*.
PhD thesis, 7 2011.

[4] Y. Yuyama, M. Aramoto, K. Kobayashi, and H. Onodera, "Rtl/iss co-modeling methodology for embedded processor using systemc.," pp. 305–308, 01 2004.

[5] N. Fournel and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible mpsoc simulation," pp. 71–80, 10 2009.

[6] "Ovpsim simulator | open virtual platforms." https://www.ovpworld.org/technology_ovpsim.
(Accessed on 03/16/2021).

[7] N. Fournel and F. Pétrot, "Using binary translation in event driven simulation for fast and flexible mpsoc simulation," pp. 71–80, 10 2009.

[8] L. Diaz, E. Gonzalez, E. Villar, and P. Sanchez, "Vippe, parallel simulation and performance analysis of multi-core embedded systems on multi-core platforms," in *Design of Circuits and Integrated Systems*, pp. 1–7, 2014.

[9] "Vippe_manual_3_1.pdf." `https://vippe.unican.es/wp-content/uploads/2020/06/VIPPE_Manual_3_1.pdf`.
(Accessed on 05/04/2021).

[10] "The llvm compiler infrastructure project." `https://llvm.org/`.
(Accessed on 03/11/2021).

[11] "Clang c language family frontend for llvm." `https://clang.llvm.org/`.
(Accessed on 03/11/2021).

[12] L. Díaz, E. González, E. Villar, and P. Sanchez, "Vippe: Native simulation and performance analysis framework for multi-processing embedded systems," 09 2014.

[13] "Documentation - ros wiki." `http://wiki.ros.org/`.
(Accessed on 03/16/2021).

[14] T. Blochwitz, M. Otter, J. Åkesson, M. Arnold, C. Clauss, H. Elmqvist, M. Friedrich, A. Junghanns, J. Mauss, D. Neumerkel, H. Olsson, and A. Viel, "Functional mockup interface 2.0: The standard for tool independent exchange of simulation models," in *Proceedings of the 9th International Modelica Conference*, pp. 173–184, The Modelica Association, 2012.

[15] S. H. A. Wittek, M. Göttsche, A. Rausch, and J. Grabowski, "Towards multi-level-simulation using dynamic cloud environments," in *2016 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH)*, pp. 1–7, 2016.

[16] M. Tranchero and L. M. Reyneri, "A multi-level simulation approach in a simulink-based design tool for fpgas," in *2009 IEEE International SOC Conference (SOCC)*, pp. 19–22, 2009.

[17] "Real-time simulation, simulation-based monitoring and predictive maintenance - fraunhofer itwm." `https://www.itwm.fraunhofer.de/en/departments/mf/dynamics-system-simulation/real-time-simulation-simulation-based-monitoring.html`.
(Accessed on 05/04/2021).

[18] "mavros - ros wiki." http://wiki.ros.org/mavros.
(Accessed on 05/12/2021).

[19] "S3d – model-driven analysis and design framework." https://s3d.unican.es/.
(Accessed on 06/24/2021).